

NOVA EDIÇÃO ATUALIZADA



O CD inclui as centenas de exemplos de 'código ativo' (live-code) em C++ do livro, links para compiladores e ferramentas de desenvolvimento em C++ e centenas de recursos na Web, incluindo referências gerais, tutoriais, FAQs, newsgroups e informações sobre STL.

C++ COMO PROGRAMAR

— 5ª Edição —

Antecipando
INTRODUÇÃO
A CLASSES
+ OBJETOS
COM UML™ 2

RING!



INTRODUÇÃO ANTECIPADA A CLASSES/OBJETOS/POO

- CLASSES, OBJETOS, ENCAPSULAMENTO
- HERANÇA, POLIMORFISMO
- ESTUDOS DE CASO POO INTEGRADOS:
TIME, GRADEBOOK, EMPLOYEE

FUNDAMENTOS

- HISTÓRIA, HARDWARE, SOFTWARE
- E/S DE FLUXO, TIPOS, OPERADORES
- INSTRUÇÕES DE CONTROLE, FUNÇÕES
- ARRAYS, VETORES
- PONTEIROS, REFERÊNCIAS
- CLASSE STRING, STRINGS NO ESTILO C
- SOBRECARGA DE OPERADOR
- EXCEÇÕES, ARQUIVOS
- PROGRAMAÇÃO WEB
- MANIPULAÇÃO DE BIT E CARACTERES
- DEPURADORES GNU™ C++/VISUAL C++®

ESTRUTURAS DE DADOS

- RECURSÃO, PESQUISA, CLASSIFICAÇÃO
- LISTAS, FILAS, PILHAS, ÁRVORES
- TEMPLATES
- STANDARD TEMPLATE LIBRARY:
CONTÊINERES, ITERADORES E ALGORITMOS

ESTUDO DE CASO OOD/UML™ 2 ATM (OPCIONAL)

- DETERMINAR CLASSES, ATRIBUTOS, ESTADOS,
ATIVIDADES, OPERAÇÕES, COLABORAÇÕES
- DIAGRAMAS: CASO DE USO, CLASSE, ESTADO,
ATIVIDADE, COMUNICAÇÃO, SEQUÊNCIA

DEITEL
DEITEL

PEARSON

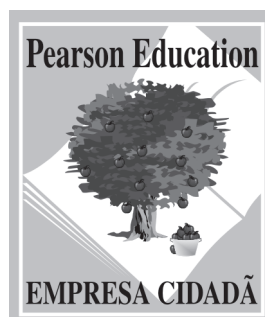
DEITEL®





C++ COMO PROGRAMAR

— 5ª Edição —



C++ COMO PROGRAMAR

— 5ª Edição —

H. M. Deitel

Deitel & Associates, Inc.

P. J. Deitel

Deitel & Associates, Inc.

Tradução

Edson Furmankiewicz

Revisão técnica

Fábio Luis Picelli Lucchini

Bacharel em Ciência da Computação pela Faculdade de Tecnologia Padre Anchieta

Mestre em Engenharia da Computação pela Unicamp (FEEC)

Professor Universitário da Faculdade de Tecnologia Padre Anchieta

PEARSON

abdr
ASSOCIAÇÃO
BRASILEIRA
DE DIREITOS
REPROGRÁFICOS
Respeite o direito autoral!

© 2006 by Pearson Education do Brasil

© 2006 by Pearson Education, Inc.

Tradução autorizada a partir da edição original em inglês, *C++, How to Program 5th edition* de Deitel, Harvey; Deitel, Paul, publicada pela Pearson Education, Inc., sob o selo Pearson Prentice Hall.

Todos os direitos reservados. Nenhuma parte desta publicação poderá ser reproduzida ou transmitida de qualquer modo ou por qualquer outro meio, eletrônico ou mecânico, incluindo fotocópia, gravação ou qualquer outro tipo de sistema de armazenamento e transmissão de informação, sem prévia autorização, por escrito, da Pearson Education do Brasil.

Gerente editorial: Roger Trimer

Editora sênior: Sabrina Cairo

Editora de desenvolvimento: Marileide Gomes

Editores de texto: Eugênia Pessotti

Preparação: Andrea Filatro

Revisão: Silvana Gouveia e Hebe Ester Lucas

Capa: Alexandre Mieda (sobre projeto original)

Projeto gráfico e diagramação: Figurativa Arte e Projeto Editorial

Dados Internacionais de Catalogação na Publicação (CIP)
(Câmara Brasileira do Livro, SP, Brasil)

Deitel, H.M., 1945- .
C++ : como programar / H.M. Deitel, P.J. Deitel ;
tradução Edson Furmankiewicz ; revisão técnica Fábio Lucchini.
— São Paulo: Pearson Prentice Hall, 2006.

Título original: C++ how to program

Bibliografia.

ISBN 978-85-4301-373-2

1. C++ (Linguagem de programação para computadores) I. Deitel, Paul J. II. Título.

05-9603

CDD-005.133

Índices para catálogo sistemático:

1. C++ : Linguagem de programação : Computadores :
Processamento de dados 005.133

8ª reimpressão – Abril 2015

Direitos exclusivos para a língua portuguesa cedidos à

Pearson Education do Brasil Ltda.,

uma empresa do grupo Pearson Education

Rua Nelson Francisco, 26

CEP 02712-100 – São Paulo – SP – Brasil

Fone: 11 2178-8686 – Fax: 11 2178-8688

e-mail: vendas@pearson.com

Marcas Comerciais

Borland e C++ Builder são marcas comerciais notórias ou marcas comerciais registradas da Borland.

Cygwin é uma marca comercial e uma obra com direitos autorais protegidos pela Red Hat, Inc. nos Estados Unidos e em outros países.

Dive Into é uma marca comercial registrada da Deitel & Associates, Inc.

GNU é uma marca comercial notória da Free Software Foundation.

Java e todas as marcas baseadas em Java são marcas comerciais notórias ou marcas comerciais registradas da Sun Microsystems, Inc. nos Estados Unidos e em outros países. A Pearson Education é uma empresa independente da Sun Microsystems, Inc.

Linux é uma marca comercial registrada de Linus Torvalds.

Microsoft, Microsoft® Internet Explorer e o logotipo Windows são marcas comerciais registradas ou marcas comerciais notórias da Microsoft Corporation nos Estados Unidos e/ou em outros países.

Janela de navegador Netscape © 2004 Netscape Communications Corporation. Utilizada com permissão. A Netscape Communications não autorizou, patrocinou, endossou ou aprovou esta publicação e não é responsável pelo seu conteúdo.

Object Management Group, OMG, Unified Modeling Language e UML são marcas comerciais registradas do Object Management Group, Inc.

Dedicatória

Para:

Stephen Clamage

Presidente do comitê J16, 'Programming Language C++' que é responsável pelo padrão C++;
engenheiro sênior de pessoal, Sun Microsystems, Inc., Divisão de Software.

Don Kostuch

Consultor independente

e Mike Miller

Ex-vice-presidente e ex-presidente do Core Language Working Group do comitê J16, 'Programming Language C++', engenheiro de design de software, Edison Design Group, Inc.

Por sua orientação, amizade e incansável devoção em insistir que nós fizéssemos 'a coisa certa' e por nos ajudar a fazer isso.

É um privilégio trabalhar com profissionais C++ tão competentes.

Harvey M. Deitel e Paul J. Deitel

Sumário

Prefácio	xxi
Antes de você começar	xxxix
I Introdução aos computadores, à Internet e à World Wide Web	I
I.1 Introdução	2
I.2 O que é um computador?	3
I.3 Organização do computador	3
I.4 Primeiros sistemas operacionais	4
I.5 Computação pessoal distribuída e computação cliente/servidor	4
I.6 A Internet e a World Wide Web	4
I.7 Linguagens de máquina, linguagens assembly e linguagens de alto nível	5
I.8 História do C e do C++	6
I.9 C++ Standard Library	6
I.10 História do Java	7
I.11 Fortran, COBOL, Ada e Pascal	7
I.12 Basic, Visual Basic, Visual C++, C# e .NET	8
I.13 Tendência-chave do software: tecnologia de objeto	8
I.14 Ambiente de desenvolvimento C++ típico	9
I.15 Notas sobre o C++ e este livro	11
I.16 Test-drive de um aplicativo C++	11
I.17 Estudo de caso de engenharia de software: introdução à tecnologia de objetos e à UML (obrigatório)	16
I.18 Síntese	20
I.19 Recursos na Web	20
2 Introdução à programação em C++	26
2.1 Introdução	27
2.2 Primeiro programa C++: imprimindo uma linha de texto	27
2.3 Modificando nosso primeiro programa C++	30
2.4 Outro programa C++: adicionando inteiros	31
2.5 Conceitos de memória	34
2.6 Aritmética	35
2.7 Tomada de decisão: operadores de igualdade e operadores relacionais	37
2.8 Estudo de caso de engenharia de software: examinando o documento de requisitos de ATM (opcional)	41
2.9 Síntese	48
3 Introdução a classes e objetos	55
3.1 Introdução	56
3.2 Classes, objetos, funções-membro e membros de dados	56
3.3 Visão geral dos exemplos do capítulo	57
3.4 Definindo uma classe com uma função-membro	57
3.5 Definindo uma função-membro com um parâmetro	60

3.6	Membros de dados, funções <i>set</i> e funções <i>get</i>	62
3.7	Inicializando objetos com construtores.....	67
3.8	Colocando uma classe em um arquivo separado para reusabilidade.....	70
3.9	Separando a interface da implementação.....	73
3.10	Validando dados com funções <i>set</i>	78
3.11	Estudo de caso de engenharia de software: identificando as classes no documento de requisitos do ATM (opcional)	82
3.12	Síntese.....	87
4	Instruções de controle: parte 1	93
4.1	Introdução	94
4.2	Algoritmos.....	94
4.3	Pseudocódigo.....	94
4.4	Estruturas de controle.....	95
4.5	Instrução de seleção <i>if</i>	98
4.6	A instrução de seleção dupla <i>if...else</i>	99
4.7	A instrução de repetição <i>while</i>	103
4.8	Formulando algoritmos: repetição controlada por contador.....	104
4.9	Formulando algoritmos: repetição controlada por sentinela.....	109
4.10	Formulando algoritmos: instruções de controle aninhadas	116
4.11	Operadores de atribuição	120
4.12	Operadores de incremento e decremento	121
4.13	Estudo de caso de engenharia de software: identificando atributos de classe no sistema ATM (opcional)	123
4.14	Síntese.....	126
5	Instruções de controle: parte 2	140
5.1	Introdução	141
5.2	Princípios básicos de repetição controlada por contador	141
5.3	A instrução de repetição <i>for</i>	143
5.4	Exemplos com a estrutura <i>for</i>	146
5.5	Instrução de repetição <i>do...while</i>	150
5.6	A estrutura de seleção múltipla <i>switch</i>	151
5.7	Instruções <i>break</i> e <i>continue</i>	159
5.8	Operadores lógicos	161
5.9	Confundindo operadores de igualdade (<i>==</i>) com operadores de atribuição (<i>=</i>)	164
5.10	Resumo de programação estruturada	165
5.11	Estudo de caso de engenharia de software: identificando estados e atividades dos objetos no sistema ATM (opcional)	169
5.12	Síntese.....	173
6	Funções e uma introdução à recursão	181
6.1	Introdução	182
6.2	Componentes de um programa em C++	183
6.3	Funções da biblioteca de matemática	184
6.4	Definições de funções com múltiplos parâmetros.....	185
6.5	Protótipos de funções e coerção de argumentos.....	189
6.6	Arquivos de cabeçalho da biblioteca-padrão C++	190
6.7	Estudo de caso: geração de números aleatórios.....	190
6.8	Estudo de caso: jogo de azar e introdução a <i>enum</i>	196

6.9	Classes de armazenamento	200
6.10	Regras de escopo	202
6.11	Pilha de chamadas de função e registros de ativação	204
6.12	Funções com listas de parâmetro vazias	206
6.13	Funções inline	208
6.14	Referências e parâmetros de referência	209
6.15	Argumentos-padrão	213
6.16	Operador de solução de escopo unário	215
6.17	Sobrecarga de funções	216
6.18	Templates de funções	218
6.19	Recursão	221
6.20	Exemplo que utiliza recursão: série de Fibonacci	222
6.21	Recursão <i>versus</i> iteração	226
6.22	Estudo de caso de engenharia de software: identificando operações de classe no sistema ATM (opcional)	228
6.23	Síntese	233
7	Arrays e vetores	250
7.1	Introdução	251
7.2	Arrays	251
7.3	Declarando arrays	252
7.4	Exemplos que utilizam arrays	253
7.5	Passando arrays para funções	267
7.6	Estudo de caso: classe GradeBook utilizando um array para armazenar notas	271
7.7	Pesquisando arrays com pesquisa linear	276
7.8	Classificando arrays por inserção	276
7.9	Arrays multidimensionais	279
7.10	Estudo de caso: classe GradeBook utilizando um array bidimensional	282
7.11	Introdução ao template vector da C++ Standard Library	288
7.12	Estudo de caso de engenharia de software: colaboração entre objetos no sistema ATM (opcional)	292
7.13	Síntese	297
8	Ponteiros e strings baseadas em ponteiro	311
8.1	Introdução	312
8.2	Declarações de variável ponteiro e inicialização	312
8.3	Operadores de ponteiro	313
8.4	Passando argumentos para funções por referência com ponteiros	315
8.5	Utilizando const com ponteiros	319
8.6	Classificação por seleção utilizando passagem por referência	324
8.7	Operadores sizeof	327
8.8	Expressões e aritmética de ponteiro	330
8.9	Relacionamento entre ponteiros e arrays	332
8.10	Arrays de ponteiros	335
8.11	Estudo de caso: simulação de embaralhamento e distribuição de cartas	336
8.12	Ponteiros de função	341
8.13	Introdução ao processamento de string baseada em ponteiro	345
	8.13.1 Fundamentos de caracteres e strings baseadas em ponteiro	346
	8.13.2 Funções de manipulação de string da biblioteca de tratamento de strings	347
8.14	Síntese	354

9	Classes: um exame mais profundo, parte I	375
9.1	Introdução	376
9.2	Estudo de caso da classe Time.....	376
9.3	Escopo de classe e acesso a membros de classe.....	382
9.4	Separando a interface da implementação.....	382
9.5	Funções de acesso e funções utilitárias	384
9.6	Estudo de caso da classe Time: construtores com argumentos-padrão	386
9.7	Destrutores	391
9.8	Quando construtores e destrutores são chamados.....	392
9.9	Estudo de caso da classe Time: uma armadilha sutil — retornar uma referência a um membro de dados private.....	395
9.10	Atribuição-padrão de membro a membro	397
9.11	Reusabilidade de software.....	399
9.12	Estudo de caso de engenharia de software: começando a programar as classes do sistema ATM (opcional)	399
9.13	Síntese	405
10	Classes: um exame mais profundo, parte 2	411
10.1	Introdução	412
10.2	Objetos const (constante) e funções-membro const	412
10.3	Composição: objetos como membros de classes	421
10.4	Funções friend e classes friend.....	426
10.5	Utilizando o ponteiro this	429
10.6	Gerenciamento de memória dinâmico com os operadores new e delete.....	430
10.7	Membros de classe static.....	435
10.8	Abstração de dados e ocultamento de informações.....	440
10.8.1	Exemplo: tipo de dados abstrato array	441
10.8.2	Exemplo: tipo de dados abstrato string.....	441
10.8.3	Exemplo: tipo de dados abstrato queue	442
10.9	Classes contêineres e iteradores.....	442
10.10	Classes proxy	442
10.11	Síntese	445
11	Sobrecarga de operadores; objetos string e array.....	450
11.1	Introdução	451
11.2	Fundamentos de sobrecarga de operadores.....	451
11.3	Restrições à sobrecarga de operadores	452
11.4	Funções operadoras como membros de classe <i>versus</i> funções globais.....	454
11.5	Sobrecarregando operadores de inserção e extração de fluxo	455
11.6	Sobrecarregando operadores unários	457
11.7	Sobrecarregando operadores binários	458
11.8	Estudo de caso: classe Array	458
11.9	Convertendo entre tipos	468
11.10	Estudo de caso: classe String	469
11.11	Sobrecarregando ++ e --	479
11.12	Estudo de caso: uma classe Date	480
11.13	Classe string da biblioteca-padrão	484
11.14	Construtores explicit.....	487
11.15	Síntese	490

12	Programação orientada a objetos: herança	501
12.1	Introdução	502
12.2	Classes básicas e derivadas	503
12.3	Membros protected	505
12.4	Relacionamento entre classes básicas e derivadas	505
12.4.1	Criando e utilizando uma classe <code>CommissionEmployee</code>	505
12.4.2	Criando uma classe <code>BasePlusCommissionEmployee</code> sem utilizar herança	510
12.4.3	Criando uma hierarquia de herança <code>CommissionEmployee–BasePlusCommissionEmployee</code>	515
12.4.4	Hierarquia de herança <code>CommissionEmployee–BasePlusCommissionEmployee</code> utilizando dados protected	519
12.4.5	Hierarquia de herança <code>CommissionEmployee–BasePlusCommissionEmployee</code> utilizando dados private	525
12.5	Construtores e destrutores em classes derivadas	532
12.6	Herança public, protected e private	539
12.7	Engenharia de software com herança	540
12.8	Síntese	541
13	Programação orientada a objetos: polimorfismo	545
13.1	Introdução	546
13.2	Exemplos de polimorfismo	547
13.3	Relacionamentos entre objetos em uma hierarquia de herança	548
13.3.1	Invocando funções de classe básica a partir de objetos de classe derivada	548
13.3.2	Apointando ponteiros de classe derivada para objetos da classe básica	554
13.3.3	Chamadas de função-membro de classe derivada via ponteiros de classe básica	555
13.3.4	Funções virtuais	557
13.3.5	Resumo das atribuições permitidas entre objetos de classe básica e de classe derivada e ponteiros	562
13.4	Campos de tipo e instruções switch	563
13.5	Classes abstratas e funções virtual puras	563
13.6	Estudo de caso: sistema de folha de pagamento utilizando polimorfismo	564
13.6.1	Criando a classe básica abstrata <code>Employee</code>	565
13.6.2	Criando a classe derivada concreta <code>SalariedEmployee</code>	566
13.6.3	Criando a classe derivada concreta <code>HourlyEmployee</code>	570
13.6.4	Criando a classe derivada concreta <code>CommissionEmployee</code>	570
13.6.5	Criando a classe derivada concreta indireta <code>BasePlusCommissionEmployee</code>	574
13.6.6	Demonstrando o processamento polimórfico	574
13.7	Polimorfismo, funções Virtual e vinculação dinâmica ‘sob o capô’ (opcional)	579
13.8	Estudo de caso: sistema de folha de pagamento utilizando polimorfismo e informações de tipo de tempo de execução com downcasting, <code>dynamic_cast</code> , <code>typeid</code> e <code>type_info</code>	582
13.9	Destrutores virtuais	585
13.10	Estudo de caso de engenharia de software: incorporando herança ao sistema ATM (opcional)	585
13.11	Síntese	591
14	Templates	595
14.1	Introdução	596
14.2	Templates de funções	596
14.3	Sobrecarregando templates de função	599
14.4	Templates de classe	599
14.5	Parâmetros sem tipo e tipos-padrão para templates de classes	605

14.6	Notas sobre templates e herança	606
14.7	Notas sobre templates e friends	606
14.8	Notas sobre templates e membros static	606
14.9	Síntese	607
15	Entrada/saída de fluxo	611
15.1	Introdução	612
15.2	Fluxos	613
15.2.1	Fluxos clássicos <i>versus</i> fluxos-padrão	613
15.2.2	Arquivos de cabeçalho da biblioteca <code>iostream</code>	613
15.2.3	Classes de entrada/saída de fluxo e objetos	614
15.3	Saída de fluxo	615
15.3.1	Saída de variáveis <code>char *</code>	616
15.3.2	Saída de caractere utilizando a função-membro <code>put</code>	616
15.4	Entrada de fluxo	616
15.4.1	Funções-membro <code>get</code> e <code>getline</code>	617
15.4.2	Funções-membro <code>peek</code> , <code>putback</code> e <code>ignore</code> de <code>istream</code>	619
15.4.3	E/S fortemente tipada (<i>type-safe</i>)	619
15.5	E/S não formatada utilizando <code>read</code> , <code>write</code> e <code>gcount</code>	620
15.6	Introdução aos manipuladores de fluxos	620
15.6.1	Base de fluxo integral: <code>dec</code> , <code>oct</code> , <code>hex</code> e <code>setbase</code>	621
15.6.2	Precisão de ponto flutuante (<code>precision</code> , <code>setprecision</code>)	621
15.6.3	Largura de campo (<code>width</code> , <code>setw</code>)	622
15.6.4	Manipuladores de fluxo de saída definidos pelo usuário	624
15.7	Estados de formato de fluxo e manipuladores de fluxo	624
15.7.1	Zeros finais e pontos de fração decimal (<code>showpoint</code>)	626
15.7.2	Alinhamento (<code>left</code> , <code>right</code> e <code>internal</code>)	626
15.7.3	Preenchimento (<code>fill</code> , <code>setfill</code>)	628
15.7.4	Base de fluxo integral (<code>dec</code> , <code>oct</code> , <code>hex</code> , <code>showbase</code>)	630
15.7.5	Números de ponto flutuante; notação científica e fixa (<code>scientific</code> , <code>fixed</code>)	631
15.7.6	Controle de letras maiúsculas/minúsculas (<code>uppercase</code>)	631
15.7.7	Especificando o formato booleano (<code>boolalpha</code>)	632
15.7.8	Configurando e redefinindo o estado de formato via a função-membro <code>flags</code>	633
15.8	Estados de erro de fluxo	634
15.9	Associando um fluxo de saída a um fluxo de entrada	636
15.10	Síntese	636
16	Tratamento de exceções	644
16.1	Introdução	645
16.2	Visão geral do tratamento de exceções	645
16.3	Exemplo: tratando uma tentativa de divisão por zero	646
16.4	Quando utilizar o tratamento de exceções	650
16.5	Relançando uma exceção	651
16.6	Especificações de exceção	653
16.7	Processando exceções inesperadas	653
16.8	Desempilhamento de pilha	654
16.9	Construtores, destrutores e tratamento de exceções	654
16.10	Exceções e herança	654
16.11	Processando falhas <code>new</code>	656
16.12	Classe <code>auto_ptr</code> e alocação de memória dinâmica	659
16.13	Hierarquia de exceções da biblioteca-padrão	661

16.14	Outras técnicas de tratamento de erro	662
16.15	Síntese	663
17	Processamento de arquivo	668
17.1	Introdução	669
17.2	Hierarquia de dados	669
17.3	Arquivos e fluxos	671
17.4	Criando um arquivo seqüencial	671
17.5	Lendo dados de um arquivo seqüencial	674
17.6	Atualizando arquivos seqüenciais	680
17.7	Arquivos de acesso aleatório.....	680
17.8	Criando um arquivo de acesso aleatório	681
17.9	Gravando dados aleatoriamente em um arquivo de acesso aleatório.....	685
17.10	Lendo um arquivo de acesso aleatório seqüencialmente	687
17.11	Estudo de caso: um programa de processamento de transação.....	689
17.12	Entrada/saída de objetos	695
17.13	Síntese	696
18	Classe string e processamento de fluxo de string.....	703
18.1	Introdução	704
18.2	Atribuição e concatenação de strings	705
18.3	Comparando strings	707
18.4	Substrings.....	709
18.5	Trocando strings	709
18.6	Características de string	710
18.7	Localizando strings e caracteres em uma string	712
18.8	Substituindo caracteres em uma string	714
18.9	Inserindo caracteres em uma string	716
18.10	Conversão para strings char * baseadas em ponteiro no estilo C	717
18.11	Iteradores.....	718
18.12	Processamento de fluxo de string	719
18.13	Síntese	722
19	Programação Web	727
19.1	Introdução	728
19.2	Tipos de solicitação HTTP	728
19.3	Arquitetura de múltiplas camadas.....	729
19.4	Acessando servidores Web	730
19.5	Apache HTTP Server	731
19.6	Solicitando documentos XHTML	731
19.7	Introdução à CGI.....	731
19.8	Transações HTTP simples	732
19.9	Scripts CGI simples	733
19.10	Enviando entrada para um script CGI	737
19.11	Utilizando formulários XHTML para enviar entrada	741
19.12	Outros cabeçalhos	745
19.13	Estudo de caso: uma página Web interativa	748
19.14	Cookies	752
19.15	Arquivos do lado do servidor	755
19.16	Estudo de caso: carrinho de compras	761

19.17	Síntese	773
19.18	Recursos na Internet e na Web	775
20	Pesquisa e classificação	780
20.1	Introdução	781
20.2	Algoritmos de pesquisa.....	782
20.2.1	Eficiência da pesquisa linear.....	782
20.2.2	Pesquisa binária.....	783
20.3	Algoritmos de classificação	787
20.3.1	Eficiência da classificação por seleção	787
20.3.2	Eficiência da classificação por inserção.....	788
20.3.3	Classificação por intercalação (uma implementação recursiva).....	788
20.4	Síntese	793
21	Estruturas de dados.....	798
21.1	Introdução	799
21.2	Classes auto-referenciais.....	799
21.3	Alocação de memória e estruturas de dados dinâmicas	800
21.4	Listas vinculadas	800
21.5	Pilhas	813
21.6	Filas	817
21.7	Árvores	820
21.8	Síntese	828
22	Bits, caracteres, strings C e structs.....	846
22.1	Introdução	847
22.2	Definições de estrutura	847
22.3	Inicializando estruturas.....	849
22.4	Utilizando estruturas com funções	849
22.5	typedef	849
22.6	Exemplo: simulação de embaralhamento e distribuição de cartas de alto desempenho	850
22.7	Operadores de bits.....	852
22.8	Campos de bit.....	859
22.9	Biblioteca de tratamento de caractere	863
22.10	Funções de conversão de string baseada em ponteiro.....	868
22.11	Funções de pesquisa da biblioteca de tratamento de strings baseadas em ponteiro.....	873
22.12	Funções de memória da biblioteca de tratamento de strings baseadas em ponteiro	874
22.13	Síntese	881
23	Standard Template Library (STL)	890
23.1	Introdução à Standard Template Library (STL)	891
23.1.1	Introdução aos contêineres.....	893
23.1.2	Introdução aos iteradores	896
23.1.3	Introdução aos algoritmos	899
23.2	Contêineres de sequência.....	901
23.2.1	Contêiner de sequência vector	902
23.2.2	Contêiner de sequência list	908
23.2.3	Contêiner de sequência deque	911
23.3	Contêineres associativos.....	913
23.3.1	Contêiner associativo multiset	913

23.3.2	Contêiner associativo set.....	915
23.3.3	Contêiner associativo multimap	915
23.3.4	Contêiner associativo map.....	918
23.4	Adaptadores de contêiner	919
23.4.1	Adaptador stack.....	919
23.4.2	Adaptador queue.....	921
23.4.3	Adaptador priority_queue	922
23.5	Algoritmos.....	924
23.5.1	fill, fill_n, generate e generate_n.....	924
23.5.2	equal, mismatch e lexicographical_compare.....	926
23.5.3	remove, remove_if, remove_copy e remove_copy_if	927
23.5.4	replace, replace_if, replace_copy e replace_copy_if.....	930
23.5.5	Algoritmos matemáticos	932
23.5.6	Algoritmos de pesquisa e classificação básica	934
23.5.7	swap, iter_swap e swap_ranges	937
23.5.8	copy_backward, merge, unique e reverse	938
23.5.9	inplace_merge, unique_copy e reverse_copy.....	940
23.5.10	Operações set.....	940
23.5.11	lower_bound, upper_bound e equal_range	943
23.5.12	Heapsort.....	945
23.5.13	min e max.....	948
23.5.14	Algoritmos de STL não discutidos neste capítulo	948
23.6	Classe bitset	950
23.7	Objetos de função.....	953
23.8	Síntese.....	955
23.9	Recursos sobre C++ na Internet e na Web	955
24	Outros tópicos	963
24.1	Introdução	964
24.2	Operador const_cast.....	964
24.3	namespaces.....	965
24.4	Palavras-chave de operador.....	968
24.5	Membros de classe mutable.....	970
24.6	Ponteiros para membros de classe (. * e ->*)	972
24.7	Herança múltipla.....	973
24.8	Herança múltipla e classes básicas virtual	976
24.9	Síntese.....	981
24.10	Observações finais	982
A	Tabela de precedência e associatividade de operadores	986
	Precedência de operadores	986
B	Conjunto de caracteres ASCII	988
C	Tipos fundamentais	989
D	Sistemas de numeração	991
D.1	Introdução	992
D.2	Abreviando números binários como números octais e números hexadecimais.....	993
D.3	Convertendo números octais e hexadecimais em números binários.....	995

D.4	Convertendo de octal, binário ou hexadecimal para decimal.....	995
D.5	Convertendo de decimal para octal, binário ou hexadecimal.....	996
D.6	Números binários negativos: notação de complemento de dois	997
E	Tópicos sobre o código C legado.....	1001
E.1	Introdução	1002
E.2	Redirecionando entrada/saída em sistemas UNIX/Linux/Mac OS X e Windows	1002
E.3	Listas de argumentos de comprimento variável	1003
E.4	Utilizando argumentos de linha de comando	1005
E.5	Notas sobre a compilação de programas de arquivo de múltiplas fontes.....	1005
E.6	Terminação de programa com <code>exit</code> e <code>atexit</code>	1007
E.7	O qualificador de tipo <code>volatile</code>	1009
E.8	Sufixos para constantes de inteiro e de ponto flutuante	1009
E.9	Tratamento de sinal	1009
E.10	Alocação dinâmica de memória com <code>calloc</code> e <code>realloc</code>	1011
E.11	O desvio incondicional: <code>goto</code>	1012
E.12	Unões	1012
E.13	Especificações de linkagem.....	1016
E.14	Síntese.....	1016
F	Pré-processador.....	1021
F.1	Introdução	1022
F.2	A diretiva de pré-processador <code>#include</code>	1022
F.3	Diretiva de pré-processador <code>#define</code> : constantes simbólicas	1022
F.4	Diretiva de pré-processador <code>#define</code> : macros	1023
F.5	Compilação condicional	1024
F.6	As diretivas de pré-processador <code>#error</code> e <code>#pragma</code>	1025
F.7	Os operadores <code>#</code> e <code>##</code>	1025
F.8	Constantes simbólicas predefinidas	1026
F.9	Assertivas	1026
F.10	Síntese.....	1026
G	Código para o estudo de caso do ATM.....	1030
G.1	Implementação do estudo de caso ATM	1030
G.2	Classe ATM.....	1031
G.3	Classe Screen.....	1037
G.4	Classe Keypad.....	1037
G.5	Classe CashDispenser.....	1038
G.6	Classe DepositSlot.....	1039
G.7	Classe Account	1040
G.8	Classe BankDatabase.....	1042
G.9	Classe Transaction.....	1045
G.10	Classe BalanceInquiry	1046
G.11	Classe Withdrawal	1048
G.12	Classe Deposit	1050
G.13	Programa de teste <code>ATMCaseStudy.cpp</code>	1055
G.14	Síntese.....	1056
H	UML 2: tipos de diagramas adicionais.....	1057
H.1	Introdução	1057

H.2	Tipos de diagramas adicionais	1057
I	Recursos sobre C++ na Internet e na Web.....	1059
I.1	Recursos	1059
I.2	Tutoriais.....	1060
I.3	FAQs.....	1061
I.4	Visual C++	1061
I.5	Newsgroups.....	1061
I.6	Compiladores e ferramentas de desenvolvimento	1061
I.7	Standard Template Library.....	1062
J	Introdução à XHTML.....	1063
J.1	Introdução	1064
J.2	Editando XHTML	1064
J.3	Primeiro exemplo de XHTML	1064
J.4	Cabeçalhos	1066
J.5	Vinculando	1068
J.6	Imagens	1070
J.7	Caracteres especiais e mais quebras de linha.....	1074
J.8	Listas não ordenadas.....	1075
J.9	Listas aninhadas e listas ordenadas.....	1077
J.10	Tabelas de XHTML básicas.....	1077
J.11	Tabelas e formatação de XHTML intermediárias.....	1081
J.12	Formulários XHTML básicos	1083
J.13	Formulários XHTML mais complexos	1086
J.14	Recursos na Internet e na Web	1092
K	Caracteres especiais de XHTML.....	1096
L	Utilizando o depurador do Visual Studio .NET.....	1097
L.1	Introdução	1098
L.2	Pontos de interrupção e o comando Continue	1098
L.3	As janelas Locals e Watch	1103
L.4	Controlando a execução utilizando os comandos Step Into , Step Over , Step Out e Continue	1105
L.5	A janela Autos	1108
L.6	Síntese	1109
M	Utilizando o depurador do GNU C++	1111
M.1	Introdução	1112
M.2	Pontos de interrupção e os comandos run , stop , continue e print	1112
M.3	Os comandos print e set	1118
M.4	Controlando a execução utilizando os comandos step , finish e next	1119
M.5	O comando watch	1121
M.6	Síntese	1124
	Bibliografia	1027
	Índice remissivo.....	1131

Prefácio

“O principal mérito da língua é a clareza ...”

Galeno

Bem-vindo ao C++ e ao C++ *Como programar, Quinta Edição*! C++ é uma linguagem de programação de primeira classe para desenvolver aplicativos de computador de capacidade industrial e de alto desempenho. Acreditamos que este livro e o material de suporte têm tudo aquilo de que os instrutores e alunos precisam para uma experiência educacional em C++ informativa, interessante, desafiadora e divertida. Neste prefácio, fornecemos uma visão geral dos muitos novos recursos do C++ *Como programar, Quinta Edição*. A seção “Passeio pelo livro” deste prefácio fornece aos instrutores, alunos e profissionais uma noção da cobertura do C++ *Como programar, Quinta Edição* sobre C++ e programação orientada a objetos. Também fornecemos uma visão geral das várias convenções utilizadas no livro, como os recursos para destaque dos exemplos de código, ‘limpeza de código’ e destaque de código. Apresentamos informações sobre compiladores livres/gratuitos que podem ser encontrados na Web. Discutimos também o abrangente conjunto de materiais educacionais que ajudam os instrutores a maximizar a experiência de aprendizagem de seus alunos (disponível no site em inglês).

Recursos do C++ *Como programar, Quinta Edição*

Na Deitel & Associates, escrevemos livros-texto de ciência da computação de nível universitário e livros profissionais. Para criar C++ *Como programar, Quinta Edição*, colocamos a edição anterior de C++ *Como programar* sob o microscópio. A nova edição tem muitos recursos atraentes:

- **Revisões importantes do conteúdo.** Todos os capítulos foram significativamente reescritos e atualizados. Adaptamos o texto para maior clareza e precisão. Também adequamos nosso uso da terminologia do C++ ao documento-padrão do ANSI/ISO C++ que define a linguagem.
- **Capítulos menores.** Os capítulos maiores foram divididos em capítulos menores, mais gerenciáveis (por exemplo, o Capítulo 1 da Quarta Edição foi dividido nos capítulos 1–2; o Capítulo 2 da Quarta Edição transformou-se nos capítulos 4–5).
- **Abordagem que introduz classes e objetos antecipadamente.** Mudamos para um método didático que introduz classes e objetos antecipadamente no currículo. Os conceitos básicos e a terminologia da tecnologia de objeto são apresentados aos alunos no Capítulo 1. Na edição anterior, os alunos começavam a desenvolver classes e objetos personalizados reutilizáveis somente no Capítulo 6, mas, nesta edição, eles fazem isso já no Capítulo 3, que é inteiramente novo. Os capítulos 4–7 foram cuidadosamente reescritos a partir de uma perspectiva de ‘introdução a classes e objetos’. Esta nova edição é orientada a objeto, onde apropriado, desde o início e por todo o texto. Deslocar a discussão sobre objetos e classes para capítulos anteriores faz com que os alunos ‘pensem sobre objetos’ imediatamente e dominem esses conceitos mais completamente. A programação orientada a objetos não é de modo algum trivial, mas é divertido escrever programas orientados a objetos, e os alunos podem ver resultados imediatos.
- **Estudos de caso integrados.** Adicionamos diversos estudos de caso distribuídos por múltiplas seções e capítulos que, com frequência, baseiam-se em uma classe introduzida anteriormente no livro para demonstrar novos conceitos de programação nos capítulos posteriores. Esses estudos de caso incluem o desenvolvimento da classe *GradeBook* nos capítulos 3–7, a classe *Time* em várias seções dos capítulos 9–10, a classe *Employee* nos capítulos 12–13 e o estudo de caso opcional ATM com OOD/UML nos capítulos 1–7, 9, 13 e Apêndice G.
- **Estudo de caso *GradeBook* integrado.** Adicionamos um novo estudo de caso *GradeBook* para reforçar nossa apresentação de classes anteriores. Ele utiliza classes e objetos nos capítulos 3–7 para construir incrementalmente uma classe *GradeBook* que representa o livro de notas de um instrutor e realiza vários cálculos com base em um conjunto de notas de alunos, como cálculo de médias, determinação das notas máximas e mínimas e impressão de um gráfico de barras.
- **Unified Modeling Language™ 2.0 (UML 2.0) — Introdução à UML 2.0.** A Unified Modeling Language (UML) tornou-se a linguagem de modelagem gráfica preferida dos projetistas de sistemas orientados a objetos. Todos os diagramas da UML neste livro obedecem à nova especificação da UML 2.0. Utilizamos diagramas de classes UML para representar visualmente classes e seus relacionamentos de herança, e usamos diagramas de atividades UML para demonstrar o fluxo de controle em cada uma das instruções de controle do C++. Fazemos uso especialmente intenso da UML no estudo de caso opcional ATM com OOD/UML.
- **Estudo de caso opcional ATM com OOD/UML.** Substituímos o estudo de caso opcional de simulador de elevador da edição anterior por um novo estudo de caso opcional de caixa eletrônico (*automated teller machine* – ATM) com OOD/UML nas seções “Estudo de caso de engenharia de software” dos capítulos 1–7, 9 e 13. O novo estudo de caso é mais simples, menor, mais ‘mundo real’ e mais apropriado aos cursos de programação de primeiro e segundo anos. As nove seções de estudo de caso apresentam uma cuidadosa introdução passo a passo ao projeto orientado a objetos utilizando a UML. Introduzimos um subconjunto simplificado e conciso da UML 2.0 e então guiamos o leitor no seu primeiro projeto concebido para projetistas e programadores iniciantes

em orientação a objetos. Nosso objetivo nesse estudo de caso é ajudar alunos a desenvolver um projeto orientado a objetos para complementar os conceitos de programação orientada a objetos que eles começam a aprender no Capítulo 1 e a implementar no Capítulo 3. O estudo de caso foi revisado por uma eminente equipe de profissionais acadêmicos e empresariais especializados em OOD/UML. O estudo de caso não é um exercício; é, mais exatamente, uma experiência de aprendizagem desenvolvida de ponta a ponta, que conclui com uma revisão detalhada da implementação de código C++ de 877 linhas. Fazemos um passeio detalhado pelas nove seções desse estudo de caso mais adiante.

- **Processo de compilação e linkagem para programas de arquivo de múltiplas fontes.** O Capítulo 3 inclui um diagrama e uma discussão detalhados do processo de compilação e linkagem que produz um aplicativo executável.
- **Explicação das pilhas de chamadas de função.** No Capítulo 6 fornecemos uma discussão detalhada (com ilustrações) sobre as pilhas de chamadas de função e sobre os registros de ativação para explicar como o C++ é capaz de monitorar a função que está atualmente em execução, como as variáveis automáticas de funções são mantidas na memória e como uma função sabe para onde retornar depois de completar a execução.
- **Introdução aos objetos `string` e `vector` da biblioteca-padrão do C++.** As classes `string` e `vector` são utilizadas para tornar os exemplos anteriores mais orientados a objetos.
- **Classe `string`.** Utilizamos a classe `string` em vez de strings `char *` baseadas em ponteiro no estilo C para a maioria das manipulações de `string` por todo o livro. Continuamos a incluir discussões sobre strings `char *` nos capítulos 8, 10, 11 e 22 para que os alunos pratiquem manipulações de ponteiro, para ilustrar a alocação dinâmica de memória com `new` e `delete`, para construir nossa própria classe `String` e para preparar alunos para atividades e projetos no mercado de trabalho nos quais eles trabalharão com strings `char *` em código C e C++ legado.
- **Template de classe `vector`.** Utilizamos o template de classe `vector` em vez de manipulações de array baseadas em ponteiro no estilo C por todo o livro. Entretanto, começamos discutindo arrays baseados em ponteiro no estilo C no Capítulo 7 para preparar os alunos para trabalhar com código C e C++ legado no mercado de trabalho e utilizá-lo como uma base para construir nossa própria classe `Array` personalizada no Capítulo 11, “Sobrecarga de operadores”.
- **Tratamento adaptado de herança e polimorfismo.** Os capítulos 12–13 foram cuidadosamente adaptados, tornando o tratamento de herança e polimorfismo mais claro e mais acessível aos alunos iniciantes em OOP. Uma hierarquia `Employee` substitui a hierarquia `Point/Circle/Cylinder` utilizada nas edições anteriores para introduzir herança e polimorfismo. A nova hierarquia é mais natural.
- **Discussão e ilustração de como o polimorfismo funciona ‘sob o capô’.** O Capítulo 13 contém um diagrama e uma explicação detalhada de como o C++ pode implementar polimorfismo, funções `virtual` e vinculação dinâmica internamente. Isso fornece aos alunos um entendimento sólido de como essas capacidades realmente funcionam. Mais importante, ajuda os alunos a apreciar o overhead do polimorfismo — em termos de consumo de memória e tempo de processador adicional. Isso ajuda os alunos a determinar quando usar polimorfismo e quando evitá-lo.
- **Programação Web.** O Capítulo 19, “Programação Web”, tem tudo aquilo de que os leitores precisam para começar a desenvolver seus próprios aplicativos baseados na Web que executarão na Internet! Os alunos aprenderão a construir os chamados aplicativos de *n*-camadas, em que a funcionalidade fornecida em cada camada pode ser distribuída para computadores separados pela Internet ou executados no mesmo computador. Utilizando o popular servidor HTTP Apache (que está disponível gratuitamente para download a partir de www.apache.org), apresentamos o protocolo CGI (Common Gateway Interface) e discutimos como a CGI permite que um servidor Web se comunique com a camada superior (por exemplo, um navegador Web que executa no computador do usuário) e com scripts CGI (isto é, nossos programas C++) que executam em um sistema remoto. Os exemplos do capítulo concluem com o estudo de caso de comércio eletrônico (*e-business*) de uma livraria on-line que permite aos usuários adicionar livros a um carrinho de compras eletrônico (*shopping cart*).
- **Standard Template Library (STL).** Este talvez seja um dos tópicos mais importantes do livro em função da avaliação que faz do reuso de software. A STL define poderosos componentes reutilizáveis, baseados em um template, que implementam muitas estruturas de dados e algoritmos usualmente utilizados para processar essas estruturas de dados. O Capítulo 23 introduz a STL e discute seus três componentes-chave — os contêineres, os iteradores e os algoritmos. Mostramos que o uso de componentes STL fornece um enorme poder expressivo e pode reduzir muitas linhas de código a uma única instrução.
- **XHTML.** O World Wide Web Consortium (W3C) declarou a Hypertext Markup Language (HTML) como uma tecnologia de legado que não sofrerá mais nenhum desenvolvimento. A HTML está sendo substituída pela Extensible Hypertext Markup Language (XHTML) — uma tecnologia baseada em XML que está se tornando rapidamente o padrão para descrever conteúdo Web. Utilizamos XHTML no Capítulo 19, “Programação Web”; o Apêndice J e o Apêndice K, introduzem a XHTML.
- **Conformidade-padrão ANSI/ISO C++.** Auditamos nossa apresentação contra o mais recente documento-padrão ANSI/ISO C++ para maior completude e exatidão. [Nota: Se você precisar de detalhes técnicos adicionais sobre C++, leia o documento-padrão C++. Uma cópia PDF eletrônica do documento-padrão C++, número INCITS/ISO/IEC 14882-2003, está disponível (em inglês) em webstore.ansi.org/ansidocstore/default.asp.]

- **Novos apêndices sobre depuradores.** Incluímos dois novos apêndices sobre utilização de depuradores — o Apêndice L, “Utilizando o depurador Visual Studio .NET”, e o Apêndice M, “Utilizando o depurador GNU C++”. Ambos estão no final do livro.
- **Novo design interno.** Reelaboramos os estilos internos da nossa série *Como programar*. As novas fontes são mais agradáveis à leitura e o novo pacote de arte é mais apropriado para as ilustrações mais detalhadas. Agora colocamos a ocorrência de definição de cada termo-chave no texto e no índice em **estilo azul negrito** para facilitar a consulta. Enfatizamos os componentes na tela com a fonte Helvetica em negrito (por exemplo, o menu Arquivo) e enfatizamos o texto do programa C++ na fonte Lucida (por exemplo, `int x = 5`).
- **Variação de tons e fontes na sintaxe.** Apresentamos o código C++ em sintaxe com algumas variações de tons e fontes. Isso melhora significativamente a legibilidade de código — um objetivo especialmente importante, já que este livro contém 17.292 linhas de código. Nossas convenções de sintaxe incluem:

```
comentários aparecem em fonte Courier
palavras-chave aparecem em azul negrito
constantes e valores literais aparecem em cinza
erros aparecem em itálico
todos os demais códigos aparecem em preto, normal
```

- **Destaque de código.** Extenso uso do destaque de código torna fácil para os leitores localizar os novos recursos de cada programa apresentado e ajuda os alunos a revisar o material rapidamente na preparação para exames ou para aulas de laboratório.
- **‘Limpeza de código’.** Este é o nosso termo para utilizar extensos comentários e identificadores significativos, aplicar convenções de recuo uniforme, alinhar chaves verticalmente, utilizar um comentário `// fim...` em cada linha com uma chave direita e utilizar o espaçamento vertical para destacar unidades de programa significativas como instruções de controle e funções. Esse processo resulta em programas fáceis de ler e autodocumentados. Fizemos a ‘limpeza de código’ de todos os programas de código-fonte tanto no texto como no material auxiliar do livro. Trabalhamos muito para tornar nosso código exemplar.
- **Teste de código em múltiplas plataformas.** Testamos os exemplos de código em várias plataformas C++ populares. Em sua maioria, todos os exemplos do livro são portados facilmente para todos os compiladores populares compatíveis com o padrão ANSI/ISO. Publicaremos qualquer problema em www.deitel.com/books/cpphttp5/index.html.
- **Erros e avisos mostrados para múltiplas plataformas.** Para os programas que contêm erros intencionais para ilustrar um conceito-chave, mostramos as mensagens de erro que resultam em várias plataformas populares.
- **Grande equipe de revisão (da edição original).** O livro foi cuidadosamente examinado por uma equipe de 30 eminentes revisores acadêmicos e da indústria (listados mais adiante no Prefácio).

Ao ler este livro, se tiver dúvidas, envie uma mensagem de correio eletrônico para deitel@deitel.com; responderemos prontamente. Visite nosso site Web, www.deitel.com, e não deixe de assinar gratuitamente nosso boletim de correio eletrônico *Buzz Online* em www.deitel.com/newsletter/subscribe.html para obter atualizações deste livro e as últimas informações sobre C++ (em inglês). Utilizamos o site Web e o boletim para manter nossos leitores e clientes corporativos informados das últimas notícias sobre publicações e serviços da Deitel. Verifique o seguinte site Web regularmente para obter erratas, atualizações relacionadas ao software C++, downloads gratuitos e outros recursos:

www.deitel.com/books/cpphttp5/index.html

A abordagem de ensino

Este livro contém uma rica coleção de exemplos, exercícios e projetos extraídos de vários campos para fornecer ao aluno uma oportunidade de resolver interessantes problemas do mundo real. O livro concentra-se nos princípios da boa engenharia de software e destaca principalmente a clareza da programação. Evitamos terminologia obscura e especificações de sintaxe em favor de ensinar por meio de exemplos. Somos educadores que ministram cursos de programação de linguagens em salas de aulas de empresas em todo o mundo. O dr. Harvey Deitel tem 20 anos de experiência no ensino universitário, incluindo a posição de chefe do Departamento de Ciência da Computação no Boston College e 15 anos de experiência no ensino em empresas. Paul Deitel tem 12 anos de experiência no ensino pedagógico em empresas. Os Deitels ministram cursos de C++ em todos os níveis para o governo, indústria, militares e clientes acadêmicos da Deitel & Associates.

Aprendendo C++ com a abordagem Live-Code (‘código ativo’)

C++ *Como programar, Quinta Edição*, é repleto de programas em C++ — cada novo conceito é apresentado no contexto de um programa em C++ funcional completo que é imediatamente seguido por uma ou mais amostras de execuções que exibem as entradas e saídas do programa. Esse estilo exemplifica a maneira como ensinamos e escrevemos sobre programação. Chamamos esse método (ensinar e escrever) de **abordagem Live-Code**. *Utilizamos as linguagens de programação para ensinar linguagens de programação*. Ler os exemplos no livro é quase como digitá-los e executá-los em um computador. Fornecemos todos os códigos-fonte para os exemplos citados no CD que acompanha e em www.deitel.com — tornando fácil para que os alunos executem cada exemplo à medida que estudam.

Acesso à World Wide Web

Todos os exemplos de código-fonte de C++ *Como programar, Quinta Edição* (e nossas outras publicações) estão disponíveis (em inglês) na Internet como downloads a partir de

www.deitel.com

O registro é rápido e fácil e os downloads são gratuitos. Sugerimos fazer download de todos os exemplos (ou copiá-los do CD que acompanha este livro) e, em seguida, executar cada programa à medida que você lê o texto correspondente. Fazer alterações nos exemplos e imediatamente ver os efeitos dessas alterações é uma excelente maneira de aprimorar sua experiência de aprendizagem em C++.

Objetivos

Cada capítulo inicia com uma declaração de objetivos. Isso permite que os alunos saibam o que esperar e oferece uma oportunidade, depois da leitura do capítulo, de verificar se eles alcançaram esses objetivos. Isso dá confiança ao aluno e é uma fonte de reforço positivo.

Citações

Os objetivos da aprendizagem são seguidos por citações. Algumas são humorísticas, algumas filosóficas e outras oferecem *insights* interessantes. Esperamos que você se divirta relacionando as citações com o material do capítulo. Muitas citações merecem uma segunda leitura após o estudo do capítulo.

Sumário

O Sumário, no início dos capítulos, oferece uma visão geral dos capítulos, de modo que os alunos possam antecipar o que virá e estabelecer um processo de aprendizagem adequado e eficiente para eles.

Aproximadamente 17.000 linhas de código apresentadas em 260 programas de exemplo com entradas e saídas de programa

Nossos programas baseados em Live-Code variam de algumas poucas linhas de código a exemplos mais substanciais. Cada programa é seguido por uma janela contendo o diálogo de entrada/saída produzido quando o programa é executado, de forma que os alunos podem confirmar que os programas são executados como o esperado. Relacionar saídas às instruções que produzem o programa é uma excelente maneira de aprender e reforçar os conceitos. Nossos programas demonstram os diversos recursos de C++. O código é numerado por linhas e apresenta sintaxe diferenciada — com palavras-chave C++, comentários e outros textos do programa, cada um com um estilo diferente. Isso facilita a leitura do código — os alunos gostarão dos estilos de sintaxe especialmente ao ler programas maiores.

735 ilustrações/figuras

Incluimos uma quantidade enorme de gráficos, tabelas, ilustrações, programas e saídas de programa. Modelamos o fluxo de controle em instruções de controle com diagramas de atividades UML. Os diagramas de classes UML modelam os membros de dados, construtores e funções-membro de classes. Utilizamos tipos adicionais de diagramas UML por todo o nosso “Estudo de caso opcional de engenharia de software ATM com OOD/UML”.

571 dicas de programação

Incluimos dicas de programação para ajudar os alunos a focalizar aspectos importantes do desenvolvimento de programa. Destacamos essas dicas na forma de *Boas práticas de programação*, *Erros comuns de programação*, *Dicas de desempenho*, *Dicas de portabilidade*, *Observações de engenharia de software* e *Dicas de prevenção de erros*. Essas dicas e práticas representam o melhor de seis décadas combinadas de programação e experiência pedagógica. Uma de nossas alunas, especializada em matemática, disse-nos que ela sente que essa abordagem é como o destaque de axiomas, teoremas, lemas e corolários em livros de matemática — fornece uma base sobre a qual criar um bom software.



Boas práticas de programação

Boas práticas de programação são dicas para escrever programas claros. Essas técnicas ajudam os alunos a criar programas mais legíveis, autodocumentados e mais fáceis de manter.



Erros comuns de programação

Alunos que começam a aprender a programação (ou uma linguagem de programação) tendem a cometer certos erros com frequência. Focalizar esses Erros comuns de programação reduz a probabilidade de os alunos cometerem os mesmos equívocos e ajuda a reduzir as longas filas que se formam na frente da sala dos instrutores fora do horário de aula!



Dicas de desempenho

Em nossa experiência, ensinar os alunos a escrever programas claros e compreensíveis é de longe o objetivo mais importante para um primeiro curso de programação. Mas os alunos querem escrever programas que executam mais rápido, utilizem menos memória,

exijam menos pressionamentos de tecla ouousem mais sobre outros aspectos. Os alunos realmente se preocupam com desempenho. Eles querem saber o que podem fazer para ‘turbinar’ seus programas. Portanto, destacamos as oportunidades de melhorar o desempenho do programa — tornando a sua execução mais rápida ou minimizando a quantidade de memória que eles ocupam.



Dicas de portabilidade

O desenvolvimento de softwares é uma atividade complexa e cara. Frequentemente, organizações que desenvolvem software devem produzir versões personalizadas para uma variedade de computadores e sistemas operacionais. Por isso, hoje há uma forte ênfase em portabilidade, isto é, na produção de softwares que executem em uma variedade de sistemas operacionais com pouca ou nenhuma alteração. Alguns programadores assumem que, se eles implementam um aplicativo em padrão C++, o aplicativo será portátil. Esse simplesmente não é o caso. Alcançar portabilidade exige um projeto cuidadoso e cauteloso. Há muitas armadilhas. Incluímos as Dicas de portabilidade para ajudar os alunos a escrever código portátil e fornecemos sugestões sobre como o C++ pode alcançar seu alto grau de portabilidade.



Observações de engenharia de software

O paradigma da programação orientada a objetos necessita de uma reconsideração completa sobre a maneira como construímos sistemas de software. O C++ é uma linguagem eficaz para alcançar a boa engenharia de software. As Observações de engenharia de software destacam questões arquitetônicas e de projeto que afetam a construção de sistemas de software, especialmente sistemas de larga escala. Muito do que o aluno aprende aqui será útil em cursos de nível superior e na indústria quando ele começar a trabalhar com sistemas grandes e complexos do mundo real.



Dicas de prevenção de erros

Quando pela primeira vez projetamos esse ‘tipo de dica’, achamos que as utilizaríamos estritamente para dizer às pessoas como testar e depurar programas C++. Na realidade, boa parte das dicas descreve os aspectos do C++ que reduzem a probabilidade de ‘bugs’, simplificando assim os processos de teste e depuração.

Seções de síntese

Cada capítulo termina com recursos pedagógicos adicionais. Nova nesta edição, cada capítulo termina com uma breve seção de ‘síntese’ que recapitula os tópicos apresentados. As sínteses também ajudam o aluno na transição para o capítulo seguinte.

Resumo (1.126 itens de resumo)

Apresentamos um resumo completo, em estilo lista de itens, no final de cada capítulo. Em média, há 40 itens de resumo por capítulo. Isso focaliza a revisão do aluno e reforça conceitos-chave.

Terminologia (1.682 termos)

Incluímos uma lista em ordem alfabética dos termos importantes definidos em cada capítulo — mais uma vez, para mais reforço. Em média, há 82 termos por capítulo. Cada termo também aparece no índice, e a ocorrência que define cada termo é destacada no índice com um número de página em **azul negrito** para permitir localizar as definições dos termos-chave rapidamente.

609 exercícios e respostas de revisão (a contagem inclui partes separadas)

Extensos exercícios de revisão e respostas são incluídos em cada capítulo. Isso oferece ao aluno uma oportunidade de adquirir confiança com o material e preparar-se para os exercícios regulares. Encorajamos os alunos a fazer todos os exercícios de revisão e a verificar suas respostas.

849 exercícios

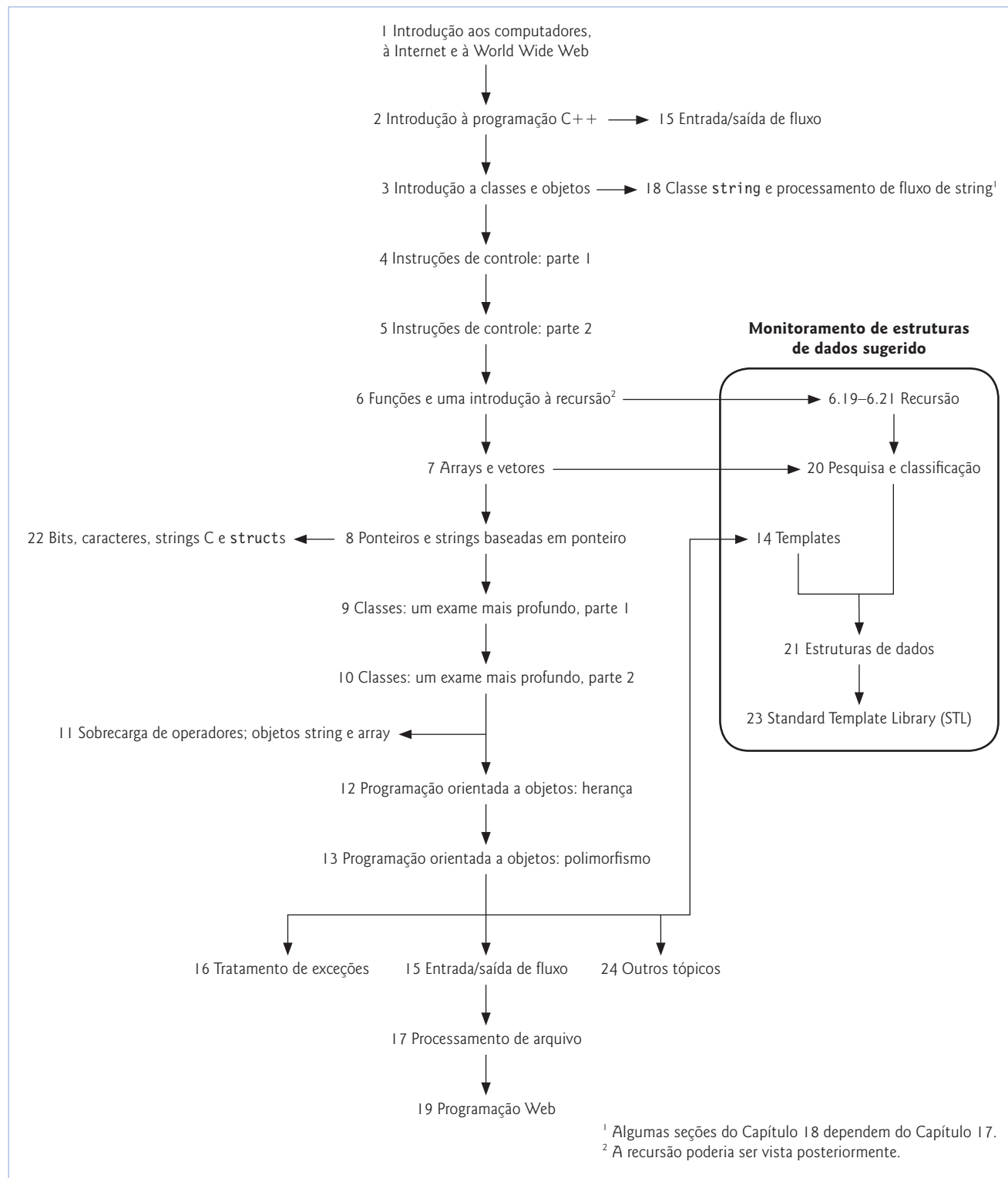
Todo capítulo conclui com um conjunto substancial de exercícios que inclui revisar terminologia e conceitos importantes; escrever instruções C++ individuais; escrever pequenas partes de funções e classes C++; escrever funções, classes e programas C++, completos; e escrever projetos de especialização da graduação. O grande número de exercícios permite que os instrutores personalizem seus cursos de acordo com as necessidades específicas de suas classes e variem as atividades de curso a cada semestre. Os instrutores podem utilizar esses exercícios para preparar deveres de casa, pequenos questionários e exames importantes. As soluções para a maioria dos exercícios estão incluídas no manual de soluções, disponível em inglês para os professores no site do livro em www.prenhall.com/deitel_br.

Aproximadamente 6.000 entradas de índice

Incluímos um extenso índice. Isso ajuda os alunos a encontrar termos ou conceitos por palavra-chave. O índice é um excelente apoio para as pessoas que estão lendo o livro pela primeira vez e é especialmente útil para programadores profissionais que utilizam o livro como uma referência.

Passeio pelo livro

Nesta seção, fazemos um passeio pelas muitas capacidades do C++ que você estudará em *C++ Como programar, Quinta Edição*. A Figura 1 ilustra as dependências entre os capítulos. Recomendamos o estudo desses tópicos na ordem indicada pelas setas, embora outras seqüências sejam possíveis. Este livro é amplamente utilizado em todos os níveis dos cursos de programação C++. Pesquise ‘syllabus’, ‘C++’ e ‘Deitel’ na Web para localizar conteúdos curriculares de disciplinas que utilizaram as recentes edições deste livro.



¹ Algumas seções do Capítulo 18 dependem do Capítulo 17.

² A recursão poderia ser vista posteriormente.

Figura 1 Fluxograma ilustrando as dependências entre os capítulos do *C++ Como Programar, Quinta Edição*.

O Capítulo 1 — Introdução aos computadores, à Internet e à Web — discute o que são computadores, como eles funcionam e como são programados. Esse capítulo fornece uma breve história do desenvolvimento das linguagens de programação desde as linguagens de máquina, passando pelas linguagens assembly, até as linguagens de alto nível. A origem da linguagem de programação C++ é discutida. Nossas publicações da série *Dive-Into™* para outras plataformas estão disponíveis (em inglês) em www.deitel.com/books/downloads.html. O capítulo inclui uma introdução para um ambiente de programação C++ típico. Guiamos os leitores por um ‘test-drive’ de um aplicativo C++ nas plataformas Windows e Linux. Esse capítulo também introduz os conceitos básicos e a terminologia da tecnologia de objetos e da UML.

O Capítulo 2 — Introdução à programação C++ — fornece uma rápida introdução à programação de aplicativos na linguagem de programação C++. O capítulo apresenta aos não-programadores os conceitos e construtos básicos de programação. Os programas desse capítulo ilustram como exibir dados na tela e como obter dados do usuário no teclado. O capítulo termina com tratamentos detalhados da tomada de decisão e operações aritméticas.

O Capítulo 3 — Introdução a classes e objetos — fornece uma introdução amigável a classes e objetos. Cuidadosamente desenvolvido, o Capítulo 3 apresenta aos alunos uma abordagem prática e fácil para a orientação a objeto desde o início. Ele foi desenvolvido sob a orientação de uma eminente equipe de revisores de universidades e empresas. Introduzimos classes, objetos, funções-membro, construtores e membros de dados utilizando uma série de exemplos simples do mundo real. Desenvolvemos uma estrutura bem projetada para organizar programas orientados a objetos em C++. Primeiro, motivamos a noção de classes com um exemplo simples. Em seguida, apresentamos uma seqüência cuidadosamente elaborada de sete programas funcionais completos para demonstrar a criação e o uso de suas próprias classes. Esses exemplos iniciam nosso **estudo de caso integrado no desenvolvimento de uma classe de livro de notas** que os instrutores podem utilizar para manter um quadro de notas de testes dos alunos. Esse estudo de caso é aprimorado nos vários capítulos seguintes, culminando com a versão apresentada no Capítulo 7, “Arrays e vetores”. O estudo de caso de classe *GradeBook* descreve como definir uma classe e como utilizá-la para criar um objeto. O estudo de caso discute como declarar e definir funções-membro para implementar os comportamentos da classe, como declarar membros de dados para implementar os atributos da classe e como chamar funções-membro de um objeto para fazê-las realizar suas tarefas. Introduzimos a classe *string* da C++ Standard Library e criamos os objetos *string* para armazenar o nome do curso que um objeto *GradeBook* representa. O Capítulo 3 explica as diferenças entre membros de dados de uma classe e variáveis locais de uma função e como utilizar um construtor para assegurar que os dados de um objeto sejam inicializados quando o objeto é criado. Mostramos como promover a reusabilidade de software separando uma definição de classe do código de cliente (por exemplo, a função *main*) que utiliza a classe. Também introduzimos outro princípio fundamental de boa engenharia de software — a separação da interface da implementação. O capítulo inclui um diagrama detalhado e uma discussão que explicam a compilação e o processo de vinculação que produz um aplicativo executável.

O Capítulo 4 — Instruções de controle: parte 1 — focaliza o processo de desenvolvimento de programa envolvido na criação de classes úteis. Esse capítulo discute como escolher uma declaração do problema e desenvolver um programa C++ funcional a partir dela, incluindo a realização de passos intermediários em pseudocódigo. O capítulo introduz algumas instruções de controle simples para tomada de decisão (*if* e *if...else*) e repetição (*while*). Examinamos a repetição controlada por contador e controlada por sentinela utilizando a **classe *GradeBook*** do Capítulo 3 e introduzimos os operadores de atribuição, incremento e decremento do C++. O capítulo inclui **duas versões aprimoradas da classe *GradeBook***, ambas baseadas na versão final do Capítulo 3. Cada uma dessas versões inclui uma função-membro que utiliza instruções de controle para calcular a média de um conjunto de notas dos alunos. Na primeira versão, o método utiliza a repetição controlada por contador para um usuário inserir 10 notas, e então determina a nota média. Na segunda versão, a função-membro utiliza repetição controlada por sentinela para inserir um número arbitrário de notas do usuário e, então, calcula a média das notas que foram inseridas. O capítulo utiliza diagramas de atividades de UML simples para mostrar o fluxo de controle por cada uma das instruções de controle.

O Capítulo 5 — Instruções de controle: parte 2 — continua a discussão sobre as instruções de controle C++ com exemplos da instrução de repetição *for*, da instrução de repetição *do...while*, da instrução de seleção *switch*, da instrução *break* e da instrução *continue*. Criamos uma **versão aprimorada da classe *GradeBook*** que utiliza uma instrução *switch* para contar o número de notas A, B, C, D e F inseridas pelo usuário. Essa versão utiliza a repetição controlada por sentinela para inserir as notas. Enquanto você lê as notas do usuário, uma função-membro modifica membros de dados que monitoram a contagem de notas em cada categoria de nota baseada em letra. Outra função-membro da classe então utiliza esses membros de dados para exibir um relatório de resumo baseado nas notas inseridas. O capítulo inclui uma discussão de operadores lógicos.

O Capítulo 6 — Funções e uma introdução à recursão — apresenta um exame mais profundo dos objetos e suas funções-membro. Discutimos as funções da biblioteca-padrão do C++ e examinamos minuciosamente como os alunos podem criar suas próprias funções. As técnicas apresentadas no Capítulo 6 são essenciais à produção de programas adequadamente organizados, especialmente os programas de grande porte que os programadores de sistema e os programadores de aplicativo podem eventualmente desenvolver em aplicações do mundo real. A estratégia ‘dividir para conquistar’ é apresentada como um meio eficaz de resolver problemas complexos dividindo-os em componentes de interação mais simples. O primeiro exemplo do capítulo no **estudo de caso da classe *GradeBook*** continua com um exemplo de uma função com múltiplos parâmetros. Os alunos apreciarão o tratamento e a simulação de números aleatórios do capítulo, e a discussão do jogo de dados, que faz uso elegante de instruções de controle. O capítulo discute os chamados ‘aprimoramentos do C++ em relação ao C’, incluindo funções *inline*, parâmetros de referência, argumentos-padrão, o operador unário de solução de escopo, sobrecarga de funções e templates de função. Apresentamos também as capacidades de chamada por valor e por referência do C++. A tabela de arquivos de cabeçalho introduz muitos dos arquivos de cabeçalho que o leitor utilizará por todo o livro. Fornecemos uma discussão detalhada (com ilustrações) sobre a pilha de chamadas de função e sobre os registros de ativação para explicar como o C++ é

capaz de monitorar quais funções estão executando atualmente, como variáveis automáticas de funções são mantidas na memória e como uma função sabe onde retornar depois de completar a execução. O capítulo então oferece uma introdução sólida à recursão e inclui uma tabela que resume os exemplos de recursão e os exercícios distribuídos por todo o restante do livro. Alguns textos deixam a recursão para um capítulo posterior no livro; achamos que este tópico é mais bem abordado gradualmente por todo o texto. A extensa coleção de exercícios no final do capítulo inclui vários problemas clássicos de recursão, incluindo as Torres de Hanói.

O Capítulo 7 — Arrays e vetores — explica como processar listas e tabelas de valores. Discutimos a estrutura de dados em arrays de itens de dados do mesmo tipo e demonstramos como os arrays facilitam as tarefas realizadas por objetos. As primeiras partes do capítulo utilizam arrays baseados em ponteiro no estilo C, que, como você verá no Capítulo 8, são, na realidade, ponteiros para o conteúdo do array na memória. Em seguida, apresentamos arrays como objetos completos com todos os recursos, na última seção do capítulo, em que introduzimos o template de classe `vector` da biblioteca-padrão — uma robusta estrutura de dados de array. O capítulo apresenta inúmeros exemplos de arrays uni e bidimensionais. Os exemplos no capítulo investigam várias manipulações de array comuns, impressão de gráficos de barras, classificação de dados e passagem de arrays para funções. O capítulo inclui as **duas seções finais do estudo de caso GradeBook**, em que utilizamos arrays para armazenar as notas dos alunos até o final da execução de um programa. Versões anteriores da classe processavam um conjunto de notas inseridas pelo usuário, mas não mantinham os valores das notas individuais em membros de dados da classe. Neste capítulo, utilizamos arrays para permitir que um objeto da classe `GradeBook` mantenha um conjunto de notas na memória, eliminando assim a necessidade de inserir repetidamente o mesmo conjunto de notas. A primeira versão da classe armazena as notas em um array unidimensional e pode produzir um relatório contendo as notas médias e as notas mínimas e máximas e um gráfico de barras que representa a distribuição das notas. A segunda versão (isto é, a versão final no estudo de caso) utiliza um array bidimensional para armazenar as notas de vários alunos de múltiplos exames em um semestre. Essa versão pode calcular a média semestral de cada aluno, bem como as notas mínimas e máximas de todas as notas recebidas no semestre. A classe também produz um gráfico de barras que exibe a distribuição total das notas para o semestre. Outro recurso-chave deste capítulo é a discussão de técnicas de classificação e pesquisa elementares. Os exercícios no final do capítulo incluem uma variedade de problemas interessantes e desafiadores, como técnicas aprimoradas de classificação, projeto de um sistema simples de reservas de passagens aéreas, uma introdução ao conceito de gráficos de tartaruga (tornados famosos na linguagem de programação LOGO) e os problemas Passeio do Cavalo e Oito Rainhas, que introduzem a noção de programação heurística amplamente empregada no campo da inteligência artificial. Os exercícios terminam com muitos problemas de recursão, que incluem classificação por seleção, palíndromos, pesquisa linear, Oito Rainhas, impressão de um array, impressão de uma string de trás para a frente e localização do valor mínimo em um array.

O Capítulo 8 — Ponteiros e strings baseadas em ponteiro — apresenta um dos recursos mais poderosos da linguagem C++ — os ponteiros. O capítulo fornece explicações detalhadas de operadores de ponteiro, chamada por referência, expressões de ponteiro, aritmética de ponteiro, relacionamento entre ponteiros e arrays, arrays de ponteiros e ponteiros para funções. Demonstramos como utilizar `const` com ponteiros para impor o princípio de menor privilégio a fim de construir um software mais robusto. Além disso, introduzimos e utilizamos o operador `sizeof` para determinar o tamanho de um tipo de dados ou itens de dados em bytes durante a compilação de programa. Há um íntimo relacionamento entre ponteiros, arrays e strings no estilo C em C++, então introduzimos conceitos de manipulação de string no estilo C básico e discutimos algumas das funções de tratamento de strings no estilo C mais populares, como `getline` (inserir uma linha de texto), `strcpy` e `strncpy` (copiar uma string), `strcat` e `strncat` (concatenar duas strings), `strcmp` e `strncmp` (comparar duas strings), `strtok` ('tokenizar' uma string em suas partes) e `strlen` (retornar o comprimento de uma string). Utilizamos os objetos `string` (introduzidos no Capítulo 3) em vez de strings `char *` baseadas em ponteiro do estilo C, onde quer que possível. Entretanto, incluímos as strings `char *` no Capítulo 8 para ajudar o leitor a dominar ponteiros e a se preparar para o mundo profissional em que ele verá muito código legado C que foi implementado nas últimas três décadas. Portanto, o leitor irá se familiarizar com os dois métodos mais predominantes de criar e manipular strings em C++. Muitas pessoas acham que o tópico ponteiros é, de longe, a parte mais difícil de um curso introdutório de programação. Em C e 'C++ bruto', arrays e strings são ponteiros para o conteúdo de array e string na memória (até mesmo nomes de função são ponteiros). Estudar esse capítulo cuidadosamente deve recompensá-lo com um profundo entendimento de ponteiros. O capítulo é repleto de exercícios desafiadores. Os exercícios incluem uma simulação da clássica competição entre a lebre e a tartaruga, algoritmos de embaralhar e distribuir cartas, classificação rápida recursiva e modos de percorrer recursivamente um labirinto. Uma seção especial intitulada "Construindo seu próprio computador" também está incluída e explica a programação de linguagem de máquina, prosseguindo com o projeto e a implementação de um simulador de computador que leva o aluno a escrever e executar programas de linguagem de máquina. Esse recurso único do texto será especialmente útil ao leitor que quer entender como os computadores realmente funcionam. Nossos alunos gostam desse projeto e freqüentemente implementam aprimoramentos substanciais, muitos dos quais são sugeridos nos exercícios. Uma segunda seção especial inclui exercícios desafiadores de manipulação de string relacionados com análise e processamento de textos, impressão de datas em vários formatos, proteção de cheque, escrita por extenso do valor de um cheque, código Morse e conversões de medidas métricas para medidas inglesas.

O Capítulo 9 — Classes: um exame mais profundo, parte 1 — continua nossa discussão sobre programação orientada a objetos. Este capítulo utiliza um rico estudo de caso da classe `Time` para ilustrar como acessar membros de classe, separar a interface da implementação, utilizar funções de acesso e funções utilitárias, inicializar objetos com construtores, destruir objetos com destrutores, atribuição por cópia-padrão de membro a membro e reusabilidade de software. Os alunos aprendem a ordem em que construtores e destrutores são chamados durante o tempo de vida de um objeto. Uma modificação no estudo de caso `Time` demonstra os problemas que podem ocorrer quando uma função-membro retorna uma referência a um membro de dados `private`, que quebra o encapsulamento da classe. Os exercícios do capítulo desafiam o aluno a desenvolver classes para horas, datas, retângulos e a jogar o jogo da velha (Tic-Tac-Toe). Em geral, os alunos gostam de programação de jogos. Leitores com inclinação para a matemática apreciarão os exercícios sobre como

criar a classe `Complex` (para números complexos), a classe `Rational` (para números racionais) e a classe `HugeInteger` (para inteiros arbitrariamente grandes).

O Capítulo 10 — Classes: um exame mais profundo, parte 2 — continua o estudo de classes e apresenta conceitos adicionais de programação orientada a objetos. O capítulo discute como declarar e utilizar objetos constantes, funções-membro constantes, composição — o processo de construir classes que têm objetos de outras classes como membros, funções `friend` e classes `friend` que têm direitos especiais de acesso aos membros das classes `private` e `protected`, o ponteiro `this`, que permite que um objeto saiba seu próprio endereço, alocação dinâmica de memória, membros de classe `static` para conter e manipular dados de escopo de classe, exemplos populares de tipos de dados abstratos (arrays, strings e filas), classes contêineres e iteradores. Em nossa discussão sobre objetos `const`, mencionamos a palavra-chave `mutable`, que é utilizada de maneira sutil para permitir a modificação da implementação ‘não visível’ nos objetos `const`. Discutimos a alocação dinâmica de memória utilizando `new` e `delete`. Quando `new` falha, o programa termina por padrão porque `new` ‘lança uma exceção’ no padrão C++. Motivamos a discussão sobre os membros da classe `static` com um exemplo baseado em videogame. Enfatizamos a importância de ocultar detalhes de implementação de clientes de uma classe; então, discutimos as classes `proxy`, que fornecem um meio de ocultar dos clientes de uma classe a implementação (incluindo os dados `private` em cabeçalhos de classe). Os exercícios de capítulo incluem o desenvolvimento de uma classe de conta-poupança e uma classe para armazenar conjuntos de inteiros.

O Capítulo 11 — Sobrecarga de operadores; objetos `string` e `array` — apresenta um dos tópicos mais populares em nossos cursos de C++. Os alunos realmente se divertem com este material. Eles o consideram uma combinação perfeita com a discussão detalhada sobre como personalizar classes valiosas nos capítulos 9 e 10. A sobrecarga de operadores permite ao programador dizer ao compilador como utilizar operadores existentes com objetos de novos tipos. O C++ já sabe utilizar esses operadores com objetos de tipos predefinidos, como inteiros, números de ponto flutuante e caracteres. Mas suponha que criássemos uma nova classe `String` — o que significaria o sinal de adição quando utilizado entre objetos `String`? Muitos programadores utilizam o sinal de adição (+) com strings para se referirem à concatenação. No Capítulo 11 o programador aprenderá a ‘sobrecarregar’ o sinal de adição, para que, quando escrito entre dois objetos `String` em uma expressão, o compilador gere uma chamada de função para uma ‘função de operador’ que irá concatenar as duas strings. O capítulo discute os princípios básicos da sobrecarga de operadores, as restrições na sobrecarga de operadores, a sobrecarga com funções-membro de classe *versus* com funções não-membro, a sobrecarga de operadores unário e binário e a conversão entre tipos. O Capítulo 11 apresenta a coleção de estudos de caso substanciais incluindo uma classe de array, uma classe `String`, uma classe de data, uma classe de inteiros arbitrariamente grandes e uma classe de números complexos (estas duas últimas aparecem com código-fonte completo nos exercícios). Alunos com aptidão para a matemática gostarão de criar a classe `polynomial` nos exercícios. Este material é diferente da maioria das linguagens e cursos de programação. A sobrecarga de operadores é um tópico complexo, mas rico. Utilizar a sobrecarga de operadores ajuda a adicionar, de modo racional, ‘acabamento’ extra a suas classes. As discussões da classe `Array` e da classe `String` são particularmente valiosas para alunos que já utilizaram a classe C++ Standard Library `string` e o template de classe `vector`, que fornecem capacidades semelhantes. Os exercícios encorajam o aluno a adicionar sobrecarga de operadores às classes `Complex`, `Rational` e `HugeInteger`, permitindo manipulação conveniente de objetos dessas classes com símbolos de operador — como na matemática — em vez de com chamadas de função, como o aluno fez nos exercícios do Capítulo 10.

O Capítulo 12 — Programação orientada a objetos: herança — introduz uma das capacidades mais fundamentais das linguagens de programação orientada a objetos — a herança: uma forma de reusabilidade de software em que novas classes são desenvolvidas rápida e facilmente absorvendo as capacidades de classes existentes e adicionando novas capacidades apropriadas. No contexto de um estudo de caso da **hierarquia `Employee`**, esse capítulo substancialmente revisado apresenta uma sequência de cinco exemplos para demonstrar dados `private`, dados `protected` e boa engenharia de software com herança. Começamos demonstrando uma classe com membros de dados `private` e funções-membro `public` para manipular esses dados. Em seguida, implementamos uma segunda classe com capacidades adicionais, duplicando intencional e tediosamente grande parte do código do primeiro exemplo. O terceiro exemplo inicia nossa discussão de herança e reuso de software — utilizamos a classe do primeiro exemplo como uma classe básica e, de modo rápido e simples, fazemos com que seus dados e funcionalidades sejam herdados por uma nova classe derivada. Esse exemplo introduz o mecanismo de herança e demonstra que uma classe derivada não pode acessar diretamente membros `private` de sua classe básica. Isso motiva nosso quarto exemplo, no qual introduzimos dados `protected` na classe básica e demonstramos que a classe derivada pode, de fato, acessar os dados `protected` herdados da classe básica. O último exemplo na sequência demonstra uma adequada engenharia de software definindo os dados da classe básica como `private` e utilizando funções-membro `public` da classe básica (que foram herdados pela classe derivada) para manipular os dados `private` da classe básica na classe derivada. O capítulo discute as noções de classes básicas e derivadas, membros `protected`, herança `public`, herança `protected`, herança `private`, classes básicas diretas e indiretas, construtores e destrutores em classes básicas e classes derivadas e engenharia de software com herança. O capítulo também compara a herança (o relacionamento ‘é um’) com a composição (o relacionamento ‘tem um’).

O Capítulo 13 — Programação orientada a objetos: polimorfismo — lida com outra capacidade fundamental da programação orientada a objetos: comportamento polimórfico. O Capítulo 13, completamente revisado, fundamenta os conceitos de herança apresentados no Capítulo 12 e focaliza os relacionamentos entre classes em uma hierarquia de classes e as poderosas capacidades de processamento que esses relacionamentos permitem. Quando muitas classes são relacionadas a uma classe básica comum por herança, cada objeto de classe derivada pode ser tratado como um objeto de classe básica. Isso permite que os programas sejam escritos de maneira simples e geral, independentemente dos tipos dos objetos específicos da classe derivada. Novos tipos de objetos podem ser tratados pelo mesmo programa, tornando os sistemas mais extensíveis. O polimorfismo permite aos programas eliminar a complexa lógica `switch` em favor da lógica ‘em linha reta’ mais simples. Um gerenciador de tela de um videogame, por exemplo, pode enviar uma mensagem `draw` para

cada objeto em uma lista vinculada de objetos para estes serem desenhados. Cada objeto sabe como se desenhar. Um objeto de uma nova classe pode ser adicionado ao programa sem modificar esse programa (contanto que esse novo objeto também saiba desenhar a si próprio). O capítulo discute os mecanismos para alcançar o comportamento polimórfico por meio de funções `virtual`. Ele distingue entre classes abstratas (a partir das quais os objetos não podem ser instanciados) e classes concretas (a partir das quais os objetos podem ser instanciados). As classes abstratas são úteis para fornecer uma interface herdável para classes por toda a hierarquia. Demonstramos classes abstratas e o comportamento polimórfico revendo a **hierarquia `Employee`** do Capítulo 12. Introduzimos uma classe básica `Employee` abstrata, a partir da qual as classes `CommissionEmployee`, `HourlyEmployee` e `SalariedEmployee` herdam diretamente e a classe `BasePlusCommissionEmployee` herda indiretamente. No passado, nossos clientes profissionais insistiam para que fornecêssemos uma explicação mais profunda que mostrasse precisamente como o polimorfismo é implementado em C++ e, portanto, precisamente em que tempos de execução e ‘custos’ de memória se incorrem ao programar com essa poderosa capacidade. Respondemos desenvolvendo uma ilustração e uma explicação precisa das *vtibles* (tabelas de função `virtual`) que o compilador C++ cria automaticamente para suportar o polimorfismo. Para concluir, introduzimos as informações de tipo de tempo de execução (*run-time type information* – RTTI) e coerção dinâmica, que permite que um programa determine um tipo de objeto em tempo de execução e, em seguida, atue nesse objeto de maneira correspondente. Utilizando RTTI e coerção dinâmica, damos um aumento de salário de 10% para empregados de um tipo específico, então calculamos os rendimentos desses empregados. Para todos os outros tipos de empregado, calculamos os lucros polimorficamente.

O Capítulo 14 — Templates — discute um dos recursos de reutilização de software mais poderosos do C++, a saber, templates. Os templates de função e de classe permitem aos programadores especificar, com um único segmento de código, um intervalo inteiro de funções sobrecarregadas relacionadas (chamadas especializações de template de função) ou um intervalo inteiro de classes relacionadas (chamadas especializações de template de classe). Essa técnica é chamada programação genérica. Os templates de função foram introduzidos no Capítulo 6. Este capítulo apresenta discussões adicionais e exemplos sobre template de função. Poderíamos escrever um único template de classe para uma classe de pilha, então fazer o C++ gerar especializações separadas de template de classe, como uma classe de ‘pilhas de ints’, uma classe de ‘pilhas de floats’, uma classe de ‘pilhas de strings’ e assim por diante. O capítulo discute como utilizar parâmetros de tipo, parâmetros sem tipo e tipos-padrão para templates de classe. Também discutimos os relacionamentos entre templates e outros recursos C++, como sobrecarga, herança, membros `friend` e `static`. Os exercícios desafiam o aluno a escrever uma variedade de templates de função e templates de classe e empregar esses templates em programas completos. Aprimoramos significativamente o tratamento de templates em nossa discussão sobre contêineres, iteradores e algoritmos da Standard Template Library (STL) no Capítulo 23.

O Capítulo 15 — Entrada/saída de fluxo — contém um tratamento abrangente das capacidades de entrada/saída do C++ padrão. Este capítulo discute uma série de capacidades suficientes para realizar a maioria das operações de E/S comuns e fornece uma visão geral das capacidades restantes. Muitos dos recursos de E/S são orientados a objeto. Esse estilo de E/S utiliza outros recursos do C++, como referências, sobrecarga de funções e de operadores. As várias capacidades de E/S do C++, incluindo saída com o operador de inserção de fluxo, entrada com o operador de extração de fluxo, E/S segura para tipo, E/S formatada, E/S não formatada (para desempenho). Os usuários podem especificar como realizar E/S para objetos de tipos definidos pelo usuário sobrecarregando o operador de inserção de fluxo (<<) e o operador de extração de fluxo (>>). Essa extensibilidade é um dos recursos mais valiosos do C++. O C++ fornece vários manipuladores de fluxo que realizam tarefas de formatação. Esse capítulo discute manipuladores de fluxo que fornecem capacidades como exibir inteiros em várias bases, controlar precisão de ponto flutuante, configurar larguras de campo, exibir ponto de fração decimal e zeros finais, justificar saída, configurar e desconfigurar estado de formato e configurar o caractere de preenchimento em campos. Também apresentamos um exemplo que cria manipuladores de fluxo de saída definidos pelo usuário.

O Capítulo 16 — Tratamento de exceções — discute como o tratamento de exceções permite aos programadores escrever programas que são robustos, tolerantes a falha e apropriados para ambientes críticos à missão e ambientes críticos ao negócio. O capítulo discute quando o tratamento de exceções é apropriado; introduz as capacidades básicas do tratamento de exceções com blocos `try`, instruções `throw` e handlers `catch`; indica como e quando relançar uma exceção; explica como escrever uma especificação de exceção e processar exceções inesperadas; e discute os importantes laços entre exceções e construtores, destrutores e herança. Os exercícios nesse capítulo mostram ao aluno a diversidade e o poder das capacidades de tratamento de exceções do C++. Discutimos como relançar uma exceção e ilustramos como `new` pode falhar quando a memória esgotar. Muitos compiladores C++ mais antigos retornam 0 por padrão quando `new` falha. Mostramos o novo estilo de falha `new` lançando uma exceção `bad_alloc` (alocação ruim). Ilustramos como utilizar a função `set_new_handler` para especificar uma função personalizada a ser chamada para lidar com situações de esgotamento de memória. Discutimos como utilizar o template de classe `auto_ptr` para excluir (`delete`) dinamicamente memória alocada implicitamente, evitando, assim, vazamentos de memória. Para concluir o capítulo, apresentamos a hierarquia de exceção da Standard Library.

O Capítulo 17 — Processamento de arquivo — discute técnicas para criar e processar tanto arquivos sequenciais como de acesso aleatório. O capítulo começa com uma introdução à hierarquia de dados, de bits a bytes, campos, registros e arquivos. Em seguida, apresentamos a visão de arquivos e fluxos do C++. Discutimos arquivos sequenciais e construímos programas que mostram como abrir e fechar arquivos, como armazenar em e ler dados sequencialmente de um arquivo. Então discutimos arquivos de acesso aleatório e construímos programas que mostram como criar um arquivo para acesso aleatório, como ler e gravar dados em um arquivo com acesso aleatório e como ler dados sequencialmente de um arquivo de acesso aleatório. O estudo de caso combina as técnicas de acessar arquivos tanto sequencial como aleatoriamente em um programa completo de processamento de transações. Os alunos em nossos seminários corporativos mencionaram que, depois de estudar o material sobre processamento de arquivo, eles foram capazes de produzir programas substanciais de processamento de arquivo que se tornaram imediatamente úteis em suas empresas. Os exercícios exigem que o aluno implemente uma variedade de programas que criam e processam tanto arquivos sequenciais como arquivos de acesso aleatório.

O Capítulo 18 — Classe `string` e processamento de fluxo de `string` — discute as capacidades do C++ de inserir dados de `strings` na memória e gerar saída de dados para `strings` na memória; essas capacidades são frequentemente referidas como formatação *in-core* ou processamento de fluxo de `string`. A classe `string` é um componente necessário da Standard Library. Preservamos o tratamento de `strings` baseadas em ponteiro no estilo C no Capítulo 8 e posteriores por várias razões. Primeiro, ele reforça o entendimento do leitor sobre ponteiros. Segundo, durante a próxima década, os programadores em C++ precisarão ser capazes de ler e modificar as enormes quantidades de código C legado que acumularam no último quarto do século — esse código processa `strings` como ponteiros, assim como o faz uma grande parte do código C++ que foi escrito na indústria nos últimos anos. No Capítulo 18 discutimos atribuição, concatenação e comparação de `string`. Mostramos como determinar várias características `string` como o tamanho e a capacidade de uma `string` e se ela está ou não vazia. Discutimos como redimensionar uma `string`. Consideramos as várias funções ‘find’ que permitem localizar uma substring em uma `string` (pesquisando a `string` para a frente ou para trás) e mostramos como localizar a primeira ou a última ocorrência de um caractere selecionado de uma `string` de caracteres e como localizar a primeira ou a última ocorrência de um caractere que não está em uma `string` de caracteres selecionada. Mostramos como substituir, apagar e inserir caracteres em uma `string` e como converter um objeto `string` em uma `string char *` no estilo C.

O Capítulo 19 — Programação Web — é opcional e tem tudo o que você precisa para começar a desenvolver seus próprios aplicativos baseados na Web que realmente executarão na Internet! Você aprenderá a construir os chamados aplicativos de *n*-camadas, em que a funcionalidade fornecida em cada camada pode ser distribuída para computadores separados pela Internet ou executados no mesmo computador. Em particular, construímos um aplicativo de livreria on-line de três camadas. As informações da livreria são armazenadas na camada inferior do aplicativo, também chamada de camada de dados. Em aplicativos de capacidade industrial, a camada de dados é, em geral, um banco de dados como o Oracle, o Microsoft® SQL Server ou o MySQL. Para simplificar, utilizamos arquivos de texto e empregamos as técnicas de processamento de arquivo do Capítulo 17 para acessar e modificar esses arquivos. O usuário insere solicitações e recebe respostas na camada superior do aplicativo, também chamada de interface com o usuário ou camada cliente, que geralmente é um computador com um navegador Web popular como o Microsoft Internet Explorer, o Mac® OS X Safari™, o Mozilla Firefox, o Opera ou o Netscape®. Naturalmente, os navegadores Web sabem comunicar-se com sites Web por toda a Internet. A camada intermediária, também chamada de camada da lógica do negócio, contém um servidor Web e um programa C++ específico do aplicativo (por exemplo, nosso aplicativo de livreria). O servidor Web comunica-se com o programa C++ (e vice-versa) via protocolo CGI (Common Gateway Interface). Esse programa é referido como um script CGI. Utilizamos o servidor HTTP Apache popular, que está disponível gratuitamente para download a partir do site Web Apache, www.apache.org. Instruções de instalação do Apache para muitas plataformas populares, incluindo sistemas Linux e Windows, estão disponíveis nesse site e em www.deitel.com e www.prenhall.com/deitel. O servidor Web sabe como conversar com a camada cliente por meio da Internet utilizando um protocolo chamado HTTP (Hypertext Transfer Protocol). Discutimos os dois métodos HTTP mais populares para enviar dados para um servidor Web — GET e POST. Então discutimos o papel crucial do servidor Web em programação Web e fornecemos um exemplo simples que solicita um documento Extensible HyperText Markup Language (XHTML)¹ de um servidor Web. Discutimos CGI e como ela permite que um servidor Web se comunique com a camada superior e aplicativos CGI. Fornecemos um exemplo simples que obtém o tempo do servidor e o renderiza em um navegador. Outros exemplos demonstram como processar entrada do usuário baseada em formulário via técnicas de processamento de `string` introduzidas no Capítulo 18. Em nossos exemplos baseados em formulário utilizamos botões, campos de senha, caixas de seleção e campos de texto. Apresentamos um exemplo de portal interativo de uma empresa de viagens que exibe os preços de passagens aéreas para várias cidades. Os membros do clube de viagem podem efetuar login e visualizar os preços das passagens aéreas. Discutimos também vários métodos de armazenar dados específicos do cliente, que incluem campos ocultos (isto é, informações armazenadas em uma página Web, mas não renderizadas pelo navegador Web) e cookies — pequenos arquivos de texto que o navegador armazena na máquina do cliente. Os exemplos do capítulo terminam com um estudo de caso de uma livreria on-line que permite aos usuários adicionar livros a um carrinho de compras. Esse estudo de caso contém vários scripts CGI que interagem para formar um aplicativo completo. A livreria on-line é protegida por senha, desse modo os usuários devem primeiro efetuar login para obter acesso. Os recursos na Web do capítulo incluem informações sobre a especificação CGI, bibliotecas C++ CGI e sites Web relacionados com o servidor Apache HTTP.

O Capítulo 20 — Pesquisa e classificação — discute duas das mais importantes classes de algoritmos em ciência da computação. Consideramos uma variedade de algoritmos específicos a cada uma e os comparamos com relação ao seu consumo de memória e consumo de processador (introduzindo a notação O, que indica o grau de dificuldade que um algoritmo pode ter para resolver um problema). Pesquisar dados envolve determinar se um valor (chamado chave de pesquisa) está presente nos dados e, se estiver, encontrar a localização do valor. Nos exemplos e exercícios deste capítulo, discutimos vários algoritmos de pesquisa, incluindo: pesquisa binária e versões recursivas de pesquisa linear e binária. Pelos exemplos e exercícios, o Capítulo 20 discute a classificação por intercalação recursiva, a classificação por borbulhamento (*bubble sort*), a classificação por *bucket* e o *quicksort* recursivo.

O Capítulo 21 — Estruturas de dados — discute as técnicas utilizadas para criar e manipular estruturas de dados dinâmicas. O capítulo inicia com discussões de classes auto-referenciais e a alocação dinâmica de memória, então prossegue com uma discussão sobre como criar e manter várias estruturas de dados dinâmicas, incluindo listas vinculadas, filas, pilhas e árvores. Para cada tipo de estrutura de dados, apresentamos programas funcionais completos e mostramos saídas de exemplo. O capítulo também ajuda o aluno a dominar ponteiros. O capítulo inclui muitos exemplos que utilizam indireção e dupla indireção — um conceito particularmente difícil. Um pro-

¹ XHTML é uma linguagem de marcação para identificar os elementos de um documento XHTML (página Web) de modo que um navegador possa renderizar (isto é, exibir) essa página na tela do computador. XHTML é uma nova tecnologia projetada pelo World Wide Web Consortium para substituir a Hypertext Markup Language (HTML) como o principal meio de especificar conteúdo Web. Nos apêndices J e K introduzimos a XHTML.

blema ao trabalhar com ponteiros é que os alunos têm dificuldade em visualizar as estruturas de dados e como seus nós estão vinculados entre si. Incluímos ilustrações que mostram os links e a sequência em que são criados. O exemplo de árvore binária é uma excelente base para o estudo de ponteiros e estruturas de dados dinâmicas. Esse exemplo cria uma árvore binária, impõe eliminação de duplicatas e introduz maneiras de percorrer recursivamente uma árvore pré-ordem, na ordem e pós-ordem. Os alunos sentem-se verdadeiramente realizados quando estudam e implementam esse exemplo. Eles particularmente apreciam ver que o modo de percorrer na ordem imprime valores de nó na ordem classificada. Incluímos uma coleção substancial de exercícios. Um destaque dos exercícios é a seção especial “Construindo seu próprio compilador”. Os exercícios orientam o aluno pelo desenvolvimento de um programa de conversão de ‘infixo para pós-fixado’ e um programa de avaliação de expressão pós-fixado. Então, modificamos o algoritmo de avaliação pós-fixado para gerar código de linguagem de máquina. O compilador coloca esse código em um arquivo (utilizando as técnicas do Capítulo 17). Os alunos então executam a linguagem de máquina produzida por compiladores nos simuladores de software criados nos exercícios do Capítulo 8! Os 35 exercícios incluem pesquisar recursivamente uma lista de endereços, imprimir recursivamente uma lista de trás para a frente, excluir nó de árvore binária, percorrer uma árvore binária na ordem, imprimir árvores, escrever parte de um compilador de otimização, escrever um interpretador, inserir em/excluir de qualquer lugar em uma lista de endereços vinculada, implementar listas e filas sem ponteiros de cauda, analisar o desempenho da pesquisa e classificação de árvore binária, implementar uma classe de lista indexada e uma simulação de supermercado que utiliza sistema de enfileiramento. Depois de estudar o Capítulo 21, o leitor está preparado para o tratamento de contêineres, iteradores e algoritmos STL no Capítulo 23. Os contêineres STL são estruturas de dados pré-empacotadas organizadas em templates que a maior parte dos programadores irá considerar suficiente para a vasta maioria dos aplicativos que precisarão implementar. A STL é um grande salto para alcançar a visão de reutilização.

O Capítulo 22 — Bits, caracteres, strings C e structs — apresenta uma variedade de recursos importantes. Este capítulo começa comparando estruturas C++ com classes e, então, definindo e utilizando estruturas no estilo C. Mostramos como declarar estruturas, inicializá-las e passá-las para funções. O capítulo apresenta uma simulação de embaralhamento e distribuição de cartas de alto desempenho. É uma excelente oportunidade para o instrutor enfatizar a qualidade de algoritmos. As poderosas capacidades de manipulação de bits do C++ permitem aos programadores escrever programas que exercitam capacidades de hardware de baixo nível. Isso ajuda os programas a processar strings de bit, configurar bits individuais e armazenar informações de maneira mais compacta. Essas capacidades, muitas vezes encontradas somente em linguagens assembly de baixo nível, são estimadas por programadores que escrevem softwares de sistema, como sistemas operacionais e software de rede. Como você se lembra, introduzimos a manipulação de string `char *` no estilo C no Capítulo 8 e apresentamos funções mais populares de manipulação de string. No Capítulo 22, continuamos nossa apresentação de caracteres e strings `char *` no estilo do C. Mostramos as várias capacidades de manipulação de caractere da biblioteca `<cctype>` — como a capacidade de testar um caractere para determinar se ele é um dígito, um caractere alfabético, um caractere alfanumérico, um dígito hexadecimal, uma letra minúscula ou maiúscula. Apresentamos as demais funções de manipulação de string das várias bibliotecas relacionadas com strings; como sempre, cada função é apresentada no contexto de um programa C++ funcional e completo. Os 32 exercícios encorajam o aluno a experimentar a maioria das capacidades discutidas no capítulo. O exercício principal conduz o aluno pelo desenvolvimento de um programa de verificação ortográfica. Este capítulo apresenta um tratamento mais profundo de strings `char *` no estilo C para o benefício de programadores C++ que irão, provavelmente, trabalhar com código C legado.

O Capítulo 23 — Standard Template Library (STL) — como em todo este livro, discute a importância do reuso de software. Reconhecendo que muitas estruturas de dados e algoritmos são comumente utilizadas por programadores C++, o comitê do padrão C++ adicionou a Standard Template Library (STL) à C++ Standard Library. A STL define poderosos componentes reutilizáveis, baseados em um template que implementa muitas estruturas de dados e algoritmos comuns utilizados para processar essas estruturas de dados. A STL oferece uma prova de conceito para a programação genérica com templates — introduzida no Capítulo 14 e demonstrada em detalhes no Capítulo 21. Esse capítulo introduz a STL e discute seus três componentes-chave — contêineres (estruturas de dados populares organizadas em template), iteradores e algoritmos. Os contêineres STL são estruturas de dados capazes de armazenar objetos de qualquer tipo de dados. Veremos que há três categorias de contêineres — contêineres de primeira classe, contêineres adaptadores e semicontêineres. Os iteradores STL, que têm propriedades semelhantes às dos ponteiros, são utilizados por programas para manipular os elementos do contêiner STL. De fato, os arrays-padrão podem ser manipulados como contêineres STL, utilizando ponteiros-padrão como iteradores. Veremos que a manipulação de contêineres com iteradores é conveniente e fornece um poder expressivo quando combinada com algoritmos STL — em alguns casos, reduzindo muitas linhas de código a uma única instrução. Os algoritmos STL são funções que realizam manipulações de dados comuns como pesquisa, classificação e comparação de elementos (ou contêineres inteiros). Há, aproximadamente, 70 algoritmos implementados na STL. A maioria desses algoritmos utiliza iteradores para acessar elementos de contêiner. Veremos que cada contêiner de primeira classe suporta tipos específicos de iterador, alguns dos quais são mais poderosos que outros. O tipo de iterador suportado de um contêiner determina se o contêiner pode ou não ser utilizado com um algoritmo específico. Os iteradores encapsulam o mecanismo utilizado para acessar elementos de contêiner. Esse encapsulamento permite que muitos dos algoritmos de STL sejam aplicados a vários contêineres sem considerar a implementação subjacente do contêiner. Uma vez que os iteradores de um contêiner suportam os requisitos mínimos do algoritmo, então o algoritmo pode processar os elementos desse contêiner. Isso também permite que os programadores criem algoritmos que podem processar os elementos de múltiplos tipos diferentes de contêiner. O Capítulo 21 discute como implementar estruturas de dados com ponteiros, classes e memória dinâmica. O código baseado em ponteiro é complexo e a mais leve omissão ou descuido pode levar a sérias violações de acesso de memória e erros de vazamento de memória sem queixas do compilador. Implementar estruturas de dados adicionais como deque (filas com dupla terminação), filas de prioridade, conjuntos, mapas etc. exige considerável trabalho adicional. Além disso, se muitos programadores em um projeto grande implementarem contêineres e algoritmos semelhantes para diferentes tarefas, o código torna-se difícil de modificar, manter e depurar. Uma vantagem da STL é que

programadores podem reutilizar os contêineres, iteradores e algoritmos STL para implementar representações e manipulações de dados comuns. Esse reuso resulta em uma considerável economia de tempo de desenvolvimento e de recursos. Esse é um capítulo amigável e acessível que deve convencê-lo do valor da STL e encorajar mais investigações.

O Capítulo 24 — Outros tópicos — é uma coleção de diversos tópicos sobre o C++. Este capítulo discute um operador de coerção adicional — `const_cast`. Esse operador, junto com `static_cast` (Capítulo 5), `dynamic_cast` (Capítulo 13) e `reinterpret_cast` (Capítulo 17), fornecem um mecanismo mais robusto para conversão entre tipos do que os operadores de coerção C++ originais herdados do C (que agora são considerados obsoletos). Discutimos namespaces, um recurso particularmente crucial para desenvolvedores de softwares que criam sistemas substanciais, especialmente para aqueles que criam sistemas de bibliotecas de classes. Os namespaces impedem colisões de nome, o que pode colocar obstáculos aos esforços de criação de softwares grandes. O capítulo discute as palavras-chave de operadores, que são úteis para programadores cujos teclados não suportam certos caracteres utilizados em símbolos de operador, como `!`, `&`, `^`, `~` e `|`. Esses operadores também podem ser utilizados por programadores que não gostam de símbolos de operadores obscuros. Discutimos a palavra-chave `mutable`, que permite que um membro de um objeto `const` seja alterado. Anteriormente, isso era realizado ‘fazendo coerção da `const`-ness’, o que é considerado uma prática perigosa. Discutimos também os operadores de ‘ponteiro para membro’ `.*` e `->*`, herança múltipla (incluindo o problema de ‘herança em forma de losango’) e classes básicas `virtual`.

O Apêndice A — Tabela de precedência e associatividade de operadores — apresenta o conjunto completo de símbolos de operadores do C++, em que cada operador aparece em uma linha com seu símbolo de operador, nome e associatividade.

O Apêndice B — Conjunto de caracteres ASCII — apresenta o conjunto de caracteres ASCII, que todos os programas neste livro utilizam.

O Apêndice C — Tipos fundamentais — lista todos os tipos fundamentais definidos no *C++ Standard*.

O Apêndice D — Sistemas de numeração — discute os sistemas numéricos binário, octal, decimal e hexadecimal. Considera como converter números entre bases e explica as representações binárias do complemento de um e do complemento de dois.

O Apêndice E — Tópicos sobre código C legado — apresenta tópicos adicionais incluindo vários tópicos avançados não comumente discutidos em cursos introdutórios. Mostramos como redirecionar a entrada de um programa para vir de um arquivo, redirecionar a saída de programa para ser colocada em um arquivo, redirecionar a saída de um programa como a entrada de outro programa (*piping*) e acrescentar a saída de um programa a um arquivo existente. Desenvolvemos funções que utilizam listas de argumentos de comprimento variável e mostramos como passar argumentos de linha de comando para a função `main` e utilizá-los em um programa. Discutimos como compilar programas cujos componentes se distribuem por múltiplos arquivos, como registrar funções com `atexit` para serem executadas ao término de um programa e como terminar a execução de um programa com a função `exit`. Discutimos também os qualificadores de tipo `const` e `volatile`, a especificação do tipo de uma constante numérica utilizando os sufixos de inteiro e de ponto flutuante, a utilização da biblioteca de tratamento de sinal para interceptar eventos inesperados, a criação e utilização de arrays dinâmicos com `calloc` e `realloc`, a utilização de unions como uma técnica de economia de espaço e o uso de especificações de linkagem quando programas C++ precisam ser linkados com código C legado. Como sugere o título, esse apêndice foi elaborado principalmente para programadores em C++ que irão trabalhar com código C legado, uma vez que é quase certo que a maioria dos programadores em C++ trabalhará com isso em um determinado momento da carreira.

O Apêndice F — Pré-processador — fornece discussões detalhadas das diretivas de pré-processador. O apêndice inclui informações mais completas sobre a diretiva `#include`, que faz com que uma cópia de um arquivo especificado seja incluída no lugar da diretiva antes de o arquivo ser compilado e sobre a diretiva `#define`, que cria constantes e macros simbólicas. Explica a compilação condicional para permitir ao programador controlar a execução de diretivas de pré-processador e a compilação de código de programas. São discutidos o operador `#`, que converte seu operando em uma string, e o operador `##`, que concatena dois tokens. São apresentadas as várias constantes simbólicas predefinidas de pré-processador (`__LINE__`, `__FILE__`, `__DATE__`, `__STDC__`, `__TIME__` e `__TIMESTAMP__`). Por fim, é discutida a macro `assert` do arquivo de cabeçalho `<cassert>`, o que é valioso em testes, verificação e validação de um programa.

O Apêndice G — Código para o estudo de caso do ATM — contém a implementação de nosso estudo de caso sobre projeto orientado a objetos com a UML. Esse apêndice é discutido na visão geral do estudo de caso (apresentado a seguir).

O Apêndice H — UML 2: tipos de diagramas adicionais — fornece uma visão geral dos tipos de diagrama UML 2 que não são encontrados no estudo de caso com OOD/UML.

O Apêndice I — Recursos sobre C++ na Internet e na Web — contém uma listagem de valiosos recursos sobre C++, como demos, informações sobre compiladores populares (incluindo ‘freebies’), livros, artigos, conferências, classificados, periódicos, revistas, ajudas, tutoriais, FAQs (perguntas feitas com frequência), newsgroups, cursos baseados na Web, notícias de produtos e ferramentas de desenvolvimento C++.

O Apêndice J — Introdução à XHTML — fornece uma introdução à XHTML — uma linguagem de marcação para descrever os elementos de uma página Web de modo que um navegador, como o Microsoft Internet Explorer ou o Netscape, possa exibir essa página. O leitor deve conhecer o conteúdo desse apêndice antes de estudar o Capítulo 19, “Programação Web”. Esse apêndice não contém nenhuma programação em C++. Alguns tópicos-chave tratados nesse apêndice incluem incorporação de texto e imagens em um documento XHTML, vinculação a outros documentos XHTML, incorporação de caracteres especiais (como símbolo de direitos autorais e de marca comercial) em um documento XHTML, separação de partes de um documento XHTML com linhas horizontais, apresentação de informações em listas e tabelas e coleta de informações de usuários que navegam em um site.

O Apêndice K — Caracteres especiais de XHTML — lista muitos caracteres XHTML especiais comumente utilizados, chamados de referências de entidade de caractere.

O **Apêndice L — Utilizando o depurador do Visual Studio .NET** — demonstra recursos-chave do Visual Studio .NET Debugger, que permite que um programador monitore a execução de aplicativos para localizar e remover erros de lógica. Esse apêndice apresenta instruções passo a passo para os alunos aprenderem a utilizar o depurador de uma maneira prática.

O **Apêndice M — Utilizando o depurador do GNU C++** — demonstra recursos-chave do GNU C++ Debugger, que permite que um programador monitore a execução de aplicativos para localizar e remover erros de lógica. Esse apêndice apresenta instruções passo a passo para os alunos aprenderem a utilizar o depurador de uma maneira prática.

A **Bibliografia** lista mais de cem livros e artigos para encorajar o aluno a ler mais sobre C++ e POO.

O **Índice** abrangente permite que o leitor localize por palavra-chave qualquer termo ou conceito em todo o texto.

Projeto orientado a objetos de um ATM com a UML: um passeio pelo estudo de caso opcional de engenharia de software

Nesta seção faremos um passeio pelo estudo de caso opcional deste livro do projeto orientado a objetos com UML. Este passeio visualiza o conteúdo das nove seções de “Estudo de caso de engenharia de software” (nos capítulos 1–7, 9 e 13). Depois de completar esse estudo de caso, o leitor estará inteiramente familiarizado com um projeto e uma implementação orientados a objetos cuidadosamente revisados para um significativo aplicativo C++.

O projeto apresentado no estudo de caso ATM foi desenvolvido na Deitel & Associates, Inc. e minuciosamente examinado por uma eminente equipe de revisão acadêmica e de profissionais da indústria. Elaboramos esse projeto para atender aos requisitos de seqüências de cursos introdutórios. Sistemas ATM reais utilizados por bancos e seus clientes em todo o mundo são baseados em projetos mais sofisticados que levam em consideração muitas outras questões além das abordadas aqui. Nosso principal objetivo por todo o processo foi criar um projeto simples que fosse claro para os iniciantes em OOD e UML e, ao mesmo tempo, demonstrar conceitos-chave do OOD e técnicas de modelagem relacionadas à UML. Trabalhamos muito para manter o projeto e o código relativamente pequenos para que funcionassem bem em um curso introdutório.

A **Seção 1.17 — Estudo de caso de engenharia de software: introdução à tecnologia de objetos e à UML** — apresenta o estudo de caso do projeto orientado a objetos com a UML. A seção introduz os conceitos básicos e a terminologia da tecnologia de objetos, incluindo classes, objetos, encapsulamento, herança e polimorfismo. Discutimos a história da UML. Essa é a única seção obrigatória do estudo de caso.

A **Seção 2.8 — Estudo de caso de engenharia de software: examinando o documento de requisitos de ATM (opcional)** — discute um *documento de requisitos* para um sistema que iremos projetar e implementar — o software para um caixa eletrônico simples (*automated teller machine* – ATM). Investigamos de maneira geral a estrutura e o comportamento dos sistemas orientados a objetos. Discutimos como a UML facilitará o processo de projeto nas seções subseqüentes de “Estudo de caso de engenharia de software” fornecendo vários tipos de diagramas adicionais para modelar nosso sistema. Incluímos uma lista de URLs e referências bibliográficas sobre projetos orientados a objetos com a UML. Discutimos a interação entre o sistema ATM especificado pelo documento de requisitos e seu usuário. Especificamente, investigamos os cenários que podem ocorrer entre o usuário e o próprio sistema — estes são chamados *casos de uso*. Modelamos essas interações utilizando *diagramas de caso de uso* da UML.

A **Seção 3.11 — Estudo de caso de engenharia de software: identificando as classes no documento de requisitos do ATM (opcional)** — começa a projetar o sistema ATM. Identificamos suas classes, ou ‘blocos de construção’, extraindo os substantivos simples e substantivos compostos do documento de requisitos. Organizamos essas classes em um diagrama de classes UML que descreve a estrutura da classe da nossa simulação. O diagrama de classes também descreve relacionamentos, conhecidos como *associações*, entre classes.

A **Seção 4.13 — Estudo de caso de engenharia de software: identificando atributos de classe no sistema ATM (opcional)** — focaliza os atributos das classes discutidas na Seção 3.11. Uma classe contém *atributos* (dados) e *operações* (comportamentos). Como veremos nas seções a seguir, alterações nos atributos do objeto freqüentemente afetam o comportamento do objeto. Para determinar os atributos para as classes no nosso estudo de caso, extraímos os adjetivos que descrevem os substantivos simples e os substantivos compostos (que definiram nossas classes) do documento de requisitos, então colocamos os atributos no diagrama de classes que criamos na Seção 3.11.

A **Seção 5.11 — Estudo de caso de engenharia de software: identificando estados e atividades de objetos no sistema ATM (opcional)** — discute como um objeto, em qualquer dado momento, ocupa uma condição específica chamada *estado*. Uma *transição de estado* ocorre quando esse objeto recebe uma mensagem para alterar o estado. A UML fornece o *diagrama de estados de máquina*, que identifica o conjunto de possíveis estados que um objeto pode ocupar e modela as transições de estado desse objeto. Um objeto também tem uma *atividade* — o trabalho que ele realiza no seu tempo de vida. A UML fornece o *diagrama de atividades* — um fluxograma que modela a atividade de um objeto. Nessa seção, utilizamos os dois tipos de diagramas para começar a modelar os aspectos comportamentais específicos do nosso sistema ATM, como o ATM executa uma transação de retirada e como o ATM responde quando o usuário é autenticado.

A **Seção 6.22 — Estudo de caso de engenharia de software: identificando operações de classe no sistema ATM (opcional)** — identifica as operações, ou serviços, de nossas classes. Extraímos do documento de requisitos os verbos e frases com verbos que especificam as operações para cada classe. Modificamos então o diagrama de classes da Seção 3.11 a fim de incluir cada operação com sua classe associada. Nesse ponto no estudo de caso, teremos coletado todas as possíveis informações do documento de requisitos. Entretanto, à medida que os capítulos seguintes introduzirem tópicos como herança, modificaremos nossas classes e diagramas.

A **Seção 7.12 — Estudo de caso de engenharia de software: colaboração entre objetos no sistema ATM (opcional)** — fornece um ‘esboço’ do modelo para nosso sistema ATM. Nessa seção vemos como ele funciona. Investigamos o comportamento da simulação

discutindo *colaborações* — mensagens que objetos enviam uns para os outros para se comunicar. As operações de classe que descobrimos na Seção 6.22 se tornam as colaborações entre os objetos no nosso sistema. Determinamos as colaborações e então as reunimos em um *diagrama de comunicação* — o diagrama da UML para modelar colaborações. Esse diagrama revela quais objetos colaboram e quando. Apresentamos um diagrama de comunicação das colaborações entre objetos para realizar uma consulta de saldo no ATM. Apresentamos então um *diagrama de seqüências* da UML para modelar interações em um sistema. Esse diagrama enfatiza a ordem cronológica das mensagens. Um diagrama de seqüências modela como objetos no sistema interagem para executar transações de retirada e de depósito.

A **Seção 9.12 — Estudo de caso de engenharia de software: começando a programar as classes do sistema ATM (opcional)** — faz uma pausa no projeto do comportamento de nosso sistema. Começamos o processo de implementação para enfatizar o material discutido no Capítulo 9. Utilizando o diagrama de classes da UML da Seção 3.11, e os atributos e as operações discutidas nas seções 4.13 e 6.22, mostramos como implementar uma classe em C++ a partir de um projeto. Não implementamos todas as classes — porque ainda não completamos o processo de projeto. Trabalhando a partir de nossos diagramas da UML, criamos o código para a classe `Withdrawal`.

A **Seção 13.10 — Estudo de caso de engenharia de software: incorporando herança ao sistema ATM (opcional)** — continua nossa discussão sobre programação orientada a objetos. Examinamos a herança — classes que compartilham características comuns podem herdar atributos e operações de uma classe ‘básica’. Nessa seção, investigamos como nosso sistema ATM pode se beneficiar da utilização de herança. Documentamos nossas descobertas em um diagrama de classes que modela relacionamentos de herança — a UML denomina esses relacionamentos de *generalizações*. Modificamos o diagrama de classes da Seção 3.11 utilizando a herança para agrupar classes com características semelhantes. Esta seção conclui o projeto da parte de modelagem da nossa simulação. Implementamos esse modelo completamente em 877 linhas de código C++ no Apêndice G.

O **Apêndice G — Código para o estudo de caso do ATM** — envolve em sua maior parte projetar o modelo (isto é, os dados e a lógica) do sistema ATM. Nesse apêndice, implementamos esse modelo em C++. Utilizando todos os diagramas UML que criamos, apresentamos as classes C++ necessárias para implementar o modelo. Aplicamos os conceitos do projeto orientado a objetos com a UML e programação orientada a objetos em C++ que você aprendeu nos capítulos. Ao final do apêndice, os alunos terão completado o projeto e a implementação de um sistema prático e devem estar confiantes para abordar sistemas maiores, como aqueles que engenheiros de softwares constroem.

O **Apêndice H — UML 2: tipos de diagramas adicionais** — fornece uma visão geral dos tipos de diagrama UML 2 que não são encontrados no estudo de caso com OOD/UML.

Material complementar

A Sala Virtual deste livro (sv.pearson.com.br) oferece: para os professores — apresentações em PowerPoint em português, manual de solução e código-fonte dos exemplos do livro em inglês; e para os alunos — código-fonte dos exemplos do livro e exercícios de múltipla escolha em inglês.

O boletim de correio eletrônico gratuito *Deitel® Buzz Online*

Nosso boletim de correio eletrônico gratuito, o *Deitel® Buzz Online*, é enviado para aproximadamente 38.000 assinantes registrados e inclui comentários sobre tendências e desenvolvimentos da indústria, links para artigos gratuitos e recursos de nossos livros publicados e das próximas publicações, agendas de lançamento de produtos, erratas, desafios, relatos, informações sobre nossos cursos de treinamento corporativos e muito mais. Também é o canal para notificar nossos leitores rapidamente sobre questões relacionadas a este livro. Todas as informações estão disponíveis em inglês. Para assiná-lo, visite

www.deitel.com/newsletter/subscribe.html

Agradecimentos à equipe da edição original

Um dos grandes prazeres de escrever um livro é o reconhecimento dos esforços de muitas pessoas cujos nomes não podem aparecer na capa, mas cujo trabalho duro, cooperação, amizade e compreensão foram cruciais à produção do livro. Muitas pessoas na Deitel & Associates, Inc. dedicaram longas horas para trabalhar conosco neste projeto.

- Andrew B. Goldberg é graduado pelo Amherst College, onde obteve o grau de bacharelado em ciência da computação. Andrew atualizou os capítulos 1–13 com base na nova apresentação antecipada de classes e outras revisões de conteúdo. Ele co-projetou e co-escreveu o novo estudo de caso opcional ATM com OOD/UML que aparece nos capítulos 1–7, 9 e 13. Além disso, foi colaborador do Capítulo 19 e co-autor dos apêndices G e H.
- Jeff Listfield é graduado em ciência da computação pelo Harvard College. Jeff contribuiu para os capítulos 18, 20 e 22, os apêndices A–F e foi co-autor dos apêndices L e M.
- Su Zhang é bacharel e doutora em ciência da computação pela McGill University e contribuiu para os capítulos 14–24.
- Cheryl Yaeger graduou-se em três anos pela Boston University em ciência da computação e contribuiu para os capítulos 4, 6, 8, 9 e 13.
- Barbara Deitel, diretora financeira na Deitel & Associates, Inc., pesquisou as citações no começo de cada capítulo e fez o copidesque deste livro.

- Abbey Deitel, presidente da Deitel & Associates, Inc., é graduada em administração industrial pela Carnegie Mellon University. Ela colaborou com o Prefácio e o Capítulo 1. Fez copidesque de vários capítulos do livro, gerenciou o processo de revisão e sugeriu o tema e os nomes de bugs para a capa do livro.
- Christi Kelsey graduou-se pela Purdue University, com bacharelado em gerenciamento e especialização em sistemas de informações. Christi contribuiu para o Prefácio e o Capítulo 1. Editou o Índice, paginou o manuscrito e coordenou muitos aspectos de nosso relacionamento de publicação com a Prentice Hall.

Tivemos a felicidade de trabalhar neste projeto com uma equipe talentosa e dedicada de profissionais de publicação da Prentice Hall. Apreciamos especialmente os esforços extraordinários de nossa editora de ciência da computação, Kate Hargett, e de sua chefe e nossa mentora em publicações — Marcia Horton, editora-chefe da divisão de engenharia e de ciência da computação da Prentice Hall. Jennifer Cappello fez um trabalho extraordinário de recrutamento da equipe de revisão e gerenciamento do processo de revisão do lado da Prentice Hall. Vince O'Brien, Tom Manshreck e John Lovell fizeram um trabalho formidável como gerentes de produção deste livro. Os talentos de Paul Belfanti, Carole Anson, Xiaohong Zhu e Geoffrey Cassar ficam evidentes no projeto interno do livro e na arte da capa; e Sarah Parker gerenciou a publicação do material complementar deste livro.

Somos muito gratos pelos esforços de nossos revisores pós-publicação da quarta edição e de nossos revisores da quinta edição:

Revisores acadêmicos

Richard Albright, Goldey Beacom College
 Karen Arlien, Bismarck State College
 David Branigan, DeVry University, Illinois
 Jimmy Chen, Salt Lake Community College
 Martin Dulberg, North Carolina State University
 Ric Heishman, Northern Virginia Community College
 Richard Holladay, San Diego Mesa College
 William Honig, Loyola University
 Earl LaBatt, OPNET Technologies, Inc./ University of New Hampshire
 Brian Larson, Modesto Junior College
 Robert Myers, Florida State University
 Gavin Osborne, Saskatchewan Institute of Applied Science and Technology
 Wolfgang Pelz, The University of Akron
 Donna Reese, Mississippi State University

Revisores da indústria

Curtis Green, Boeing Integrated Defense Systems
 Mahesh Hariharan, Microsoft
 James Huddleston, Consultor independente
 Ed James-Beckham, Borland Software Corporation
 Don Kostuch, Consultor independente
 Meng Lee, Hewlett-Packard
 Kriang Lerdsuwanakij, Siemens Limited
 William Mike Miller, Edison Design Group, Inc.
 Mark Schimmel, Borland International
 Vicki Scott, Metrowerks
 James Snell, Boeing Integrated Defense Systems
 Raymond Stephenson, Microsoft

Revisores do Estudo de caso de engenharia de software com OOD/UML

Sinan Si Alhir, Consultor independente
 Karen Arlien, Bismarck State College
 David Branigan, DeVry University, Illinois
 Martin Dulberg, North Carolina State University
 Ric Heishman, Northern Virginia Community College
 Richard Holladay, San Diego Mesa College
 Earl LaBatt, OPNET Technologies, Inc./ University of New Hampshire
 Brian Larson, Modesto Junior College
 Gavin Osborne, Saskatchewan Institute of Applied Science and Technology
 Praveen Sadhu, Infodat International, Inc.
 Cameron Skinner, Embarcadero Technologies, Inc. / OMG
 Steve Tockey, Construx Software

Revisores pós-publicação de C++ 4/e

Butch Anton, Wi-Tech Consulting
 Karen Arlien, Bismarck State College
 Jimmy Chen, Salt Lake Community College
 Martin Dulberg, North Carolina State University
 William Honig, Loyola University
 Don Kostuch, Consultor independente
 Earl LaBatt, OPNET Technologies, Inc./ University of New Hampshire
 Brian Larson, Modesto Junior College
 Kriang Lerdsuwanakij, Siemens Limited
 Robert Myers, Florida State University
 Gavin Osborne, Saskatchewan Institute of Applied Science and Technology
 Wolfgang Pelz, The University of Akron
 David Papurt, Consultor Independente
 Donna Reese, Mississippi State University
 Catherine Wyman, DeVry University, Phoenix
 Salih Yurttas, Texas A&M University

Pressionados por um rígido prazo final de entrega, eles examinaram minuciosamente cada aspecto do texto e fizeram incontáveis sugestões para melhorar a exatidão e a inteireza da apresentação.

Bem, aí está! Bem-vindo ao fascinante mundo do C++ e da programação orientada a objetos. Esperamos que você goste deste exame da programação de computador contemporânea. Boa sorte! À medida que você ler este livro, apreciaríamos sinceramente seus comentários, críticas, correções e sugestões para melhorar o texto. Envie qualquer correspondência para

`clientes@pearsoned.com`

ou em inglês para

`deitel@deitel.com`

Responderemos prontamente e publicaremos correções e esclarecimentos em:

`www.deitel.com/books/cpphttp5/index.html`

Esperamos que você se divirta aprendendo com C++ *Como programar, Quinta Edição* tanto quanto nós nos divertimos ao escrevê-lo!

Dr. Harvey M. Deitel

Paul J. Deitel

Sobre os autores

O dr. **Harvey M. Deitel**, presidente e executivo-chefe de estratégias da Deitel & Associates Inc., tem 43 anos de experiência no campo de computação, incluindo extensa experiência acadêmica e corporativa. O dr. Deitel obteve os graus de B.S e M.S. do Massachusetts Institute of Technology e um Ph.D. da Boston University. Ele foi um dos pioneiros nos projetos dos sistemas operacionais de memória virtual na IBM e no MIT que desenvolveram as técnicas amplamente implementadas hoje em dia em sistemas como UNIX, Linux e Windows XP. Ele tem 20 anos de experiência no ensino universitário, incluindo livre-docência e a chefia do Departamento de Ciência da Computação no Boston College antes de fundar a Deitel & Associates, Inc., com seu filho, Paul J. Deitel. O dr. Deitel proferiu centenas de seminários profissionais para importantes corporações, instituições acadêmicas, organizações governamentais e órgãos militares. Ele e Paul são os co-autores de dezenas de livros e pacotes de multimídia e estão escrevendo muito mais. Com traduções publicadas em japonês, alemão, russo, espanhol, chinês tradicional, chinês simplificado, francês, polonês, italiano, português, grego, urdu e turco, os textos da Deitel ganharam reconhecimento internacional.

Paul J. Deitel, executivo-chefe de tecnologia da Deitel & Associates, Inc., é graduado pela Sloan School of Management do MIT, onde estudou Tecnologia da Informação. Por meio da Deitel & Associates, ministrou cursos sobre C++, Java, C, Internet e World Wide Web para clientes de grandes empresas, incluindo IBM, Sun Microsystems, Dell, Lucent Technologies, Fidelity, NASA at the Kennedy Space Center, National Severe Storm Laboratory, PalmSource, White Sands Missile Range, Rogue Wave Software, Boeing, Stratus, Cambridge Technology Partners, TJX, One Wave, Hyperion Software, Adra Systems, Entergy, CableData Systems e várias outras organizações. Paul é um dos professores corporativos mais experientes do mundo em Java e C++, tendo conduzido cerca de cem cursos sobre treinamento profissional em Java e C++. Deu palestras sobre C++ e Java para o Boston Chapter da Association for Computing Machinery. Ele e seu pai, dr. Harvey M. Deitel, são autores de livros-texto sobre a ciência da computação líderes de vendas no mundo todo.

Sobre a Deitel & Associates, Inc.

A Deitel & Associates, Inc., é uma organização internacionalmente reconhecida de treinamento corporativo e criação de conteúdo, especializada em linguagens de programação de computadores, tecnologia de software para a World Wide Web e a Internet e ensino da tecnologia de objetos. A empresa oferece cursos direcionados a instrutores sobre as principais linguagens de programação e plataformas, como Java, Advanced Java, C, C++, linguagens de programação .NET, XML, Perl, Python; tecnologia de objetos; e programação

para a Internet e a World Wide Web. Os fundadores da Deitel & Associates, Inc. são o dr. Harvey M. Deitel e Paul J. Deitel. Os clientes da empresa incluem muitas das maiores empresas de computadores do mundo, agências governamentais, setores do serviço militar e organizações comerciais. Ao longo dos seus 29 anos de parceria editorial com a Prentice Hall, a Deitel & Associates, Inc. publica livros-texto de programação de ponta, livros profissionais, cursos interativos e multimídia como os *Cyber Classrooms* e os *Complete Training Courses*, cursos de treinamento baseados na Web e conteúdo eletrônico de sistemas de gerenciamento de cursos (*course management systems* – CMSs) para CMSs populares como o WebCT, o Blackboard e o CourseCompass da Pearson. A Deitel & Associates e os autores podem ser contatados por e-mail (em inglês) em:

deitel@deitel.com

Para saber mais sobre a Deitel & Associates, suas publicações e o currículo do seu *Dive Into*™ Series Corporate Training mundial, visite o nosso site:

www.deitel.com

e assine o boletim *Deitel*® *Buzz Online* gratuito via correio eletrônico em:

www.deitel.com/newsletter/subscribe.html

Aqueles que desejam comprar livros da Deitel, *Cyber Classrooms*, *Complete Training Courses* e cursos de treinamento baseados na Web podem fazer isso via:

www.deitel.com/books/index.html

Antes de começar

Siga as instruções nesta seção para assegurar que os exemplos do livro sejam adequadamente copiados no computador antes de você começar a utilizar este livro.

Convenções de fontes e nomes

Utilizamos fontes para separar componentes na tela (como nomes de menu e itens de menu) e código ou comandos C++. Nossa convenção é enfatizar componentes de tela utilizando a fonte sem serifa Helvetica em negrito (por exemplo, menu File) e enfatizar código e comandos C++ com uma fonte Lucida sem serifa (por exemplo, `cout << "Hello";`).

Recursos no CD que acompanha C++ Como programar, Quinta Edição

O CD que acompanha este livro inclui:

- Centenas de exemplos de código ativo (Live-Code) em C++.
- Os links para compiladores C++ gratuitos e ambientes de desenvolvimento integrado (IDEs).
- Centenas de recursos na Web, incluindo referências gerais, tutoriais, FAQs, newsgroups e informações sobre a STL.

Se você tiver alguma pergunta, envie correspondência para clientes@pearsoned.com ou em inglês para deitel@deitel.com

Copiando e organizando arquivos

Todos os exemplos do C++ *Como programar, Quinta Edição* são incluídos no CD que acompanha este livro. Siga os passos na seção a seguir, “Copiando os exemplos do livro a partir do CD”, para copiar o diretório de `examples` no CD para sua unidade de disco. Sugerimos que você trabalhe a partir da sua unidade de disco em vez da sua unidade de CD por duas razões: o CD é de leitura, portanto você não pode salvar seus aplicativos no CD do livro; além disso, os arquivos podem ser acessados mais rapidamente de uma unidade de disco do que de um CD. Os exemplos no livro também estão disponíveis para download no site do livro em www.prenhall.com/deitel_br e no nosso site em:

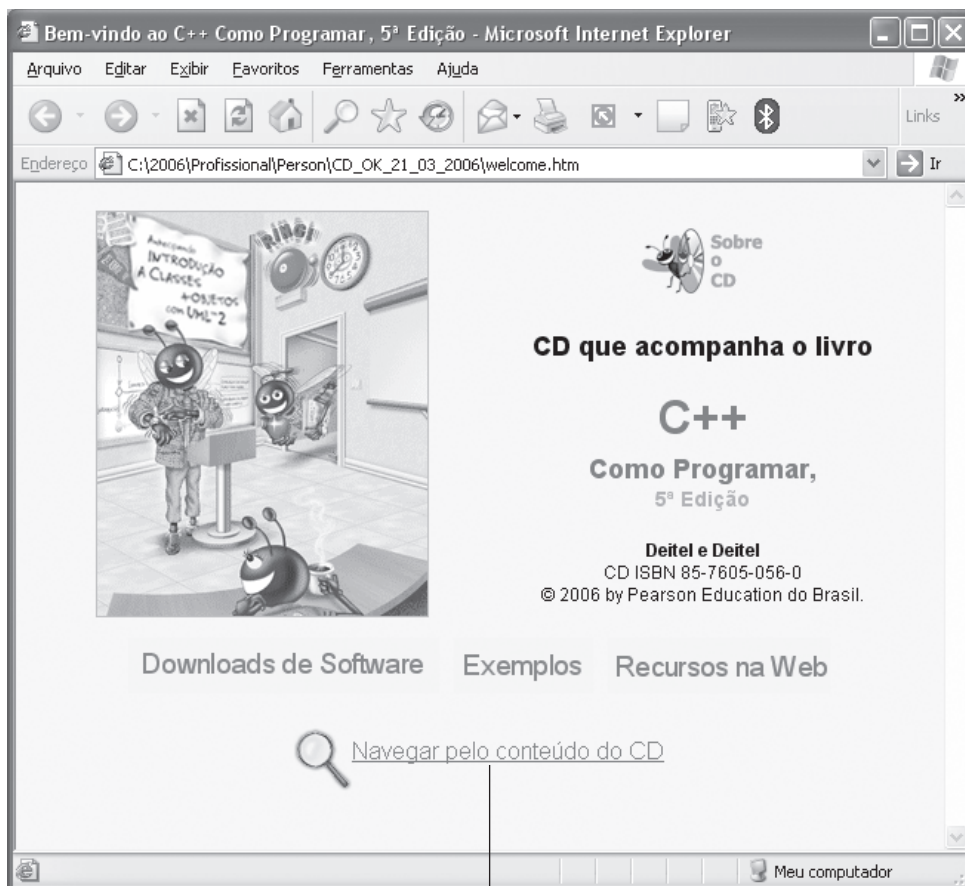
www.deitel.com/books/cpphttp5/index.html
www.prenhall.com/deitel

Pressupomos para o propósito desta seção “Antes de começar” que você esteja utilizando um computador que execute o Microsoft Windows. As capturas de tela na seção a seguir podem diferir ligeiramente do que você vê no seu computador, dependendo do uso do Windows 2000 ou Windows XP. Se você estiver executando um sistema operacional diferente e tiver perguntas sobre como copiar os arquivos de exemplo no computador, consulte seu instrutor.

Copiando os exemplos do livro a partir do CD

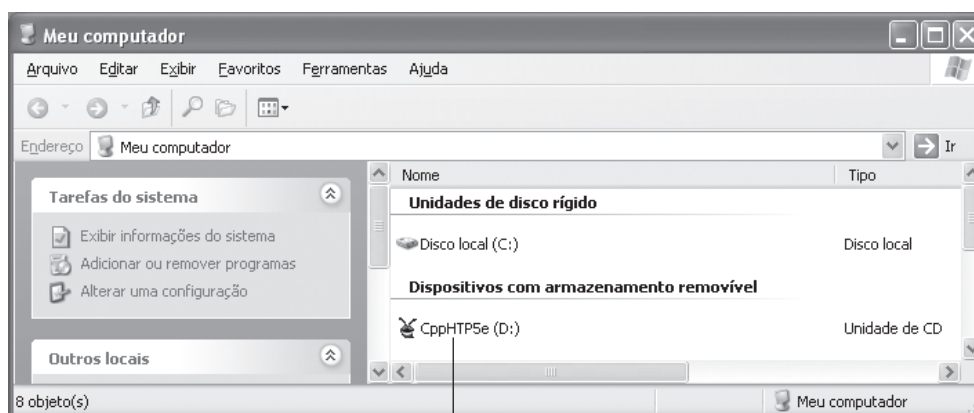
1. **Inserindo o CD.** Insira o CD que acompanha C++ *Como programar, Quinta Edição* na unidade de CD do computador. A janela exibida na Figura 1 deve aparecer. Se a página aparecer, prossiga para o *Passo 3* desta seção. Se a página não aparecer, prossiga para o *Passo 2*.
2. **Abrindo o diretório de CD utilizando Meu Computador.** Se a página mostrada na Figura 1 não aparecer, dê um clique duplo no ícone Meu Computador na área de trabalho. Na janela Meu computador, dê um clique duplo na unidade de CD-ROM (Figura 2) para carregar o CD (Figura 1).
3. **Abrindo o diretório de CD-ROM.** Se a página na Figura 1 aparecer, clique no link Navegar pelo conteúdo do CD (Figura 1) para acessar o conteúdo do CD.
4. **Copiando o diretório `examples`.** Dê um clique com o botão direito do mouse no diretório `examples` (Figura 3) e então selecione Copiar. Em seguida, vá para Meu Computador e dê um clique duplo na unidade C:. Selecione a opção Colar no menu Editar para copiar o diretório e seu conteúdo no CD para a unidade C:. [Nota: Salvamos os exemplos na unidade C: e fazemos referência a essa unidade por todo o livro. Você pode escolher salvar os arquivos em uma unidade diferente com base na configuração do seu computador, na configuração do laboratório da sua faculdade ou em suas preferências pessoais. Se você trabalha em um laboratório de informática, consulte seu instrutor para informações adicionais a fim de confirmar onde os exemplos devem ser salvos.]

Os arquivos de exemplo que você copiou para seu computador do CD são de leitura. Em seguida, você removerá a propriedade de leitura para que possa modificar e executar os exemplos.



Clique no link Navegar pelo conteúdo do CD
para acessar o conteúdo do CD

Figura 1 Página de boas-vindas do CD de C++ Como programar.



Unidade de CD-ROM selecionada

Figura 2 Localizando a unidade de CD-ROM.

Clique com o botão direito do mouse no diretório examples

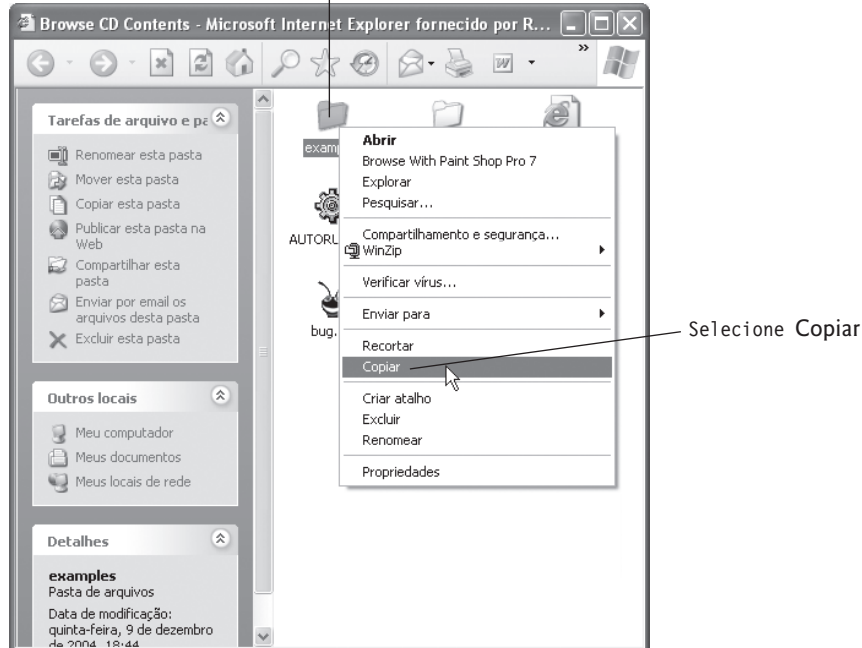


Figura 3 Copiando o diretório examples.

Alterando a propriedade de leitura dos arquivos

1. **Abrindo a caixa de diálogo *Propriedades*.** Dê um clique com o botão direito do mouse no diretório examples e selecione *Propriedades* no menu. A caixa de diálogo *Propriedades* de examples aparece (Figura 4).

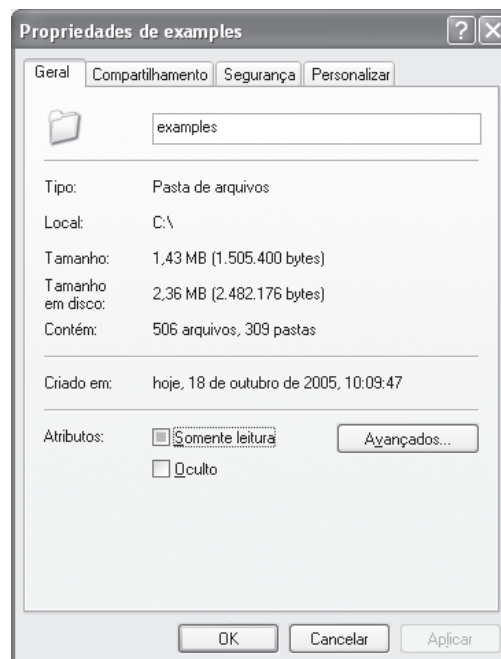


Figura 4 Caixa de diálogo *Propriedades* de examples.

2. **Alterando a propriedade *Somente leitura*.** Na seção **Atributos** dessa caixa de diálogo, clique na caixa ao lado de **Somente leitura** para remover a marca de seleção (Figura 5). Clique em **Aplicar** para aplicar as alterações.
3. **Alterando a propriedade para todos os arquivos.** Clicar em **Aplicar** exibirá a janela **Confirmar alterações de atributo** (Figura 6). Nessa janela, clique no botão de opção ao lado de **Aplicar as alterações a esta pasta, subpastas e arquivos** e clique em **OK** para remover a propriedade de somente leitura para todos os arquivos e diretórios no diretório `examples`.

Agora você está pronto para começar seus estudos sobre o C++ com *C++ Como programar*. Esperamos que você goste deste livro! Você pode nos contatar facilmente em `deitel@deitel.com`. Responderemos prontamente.

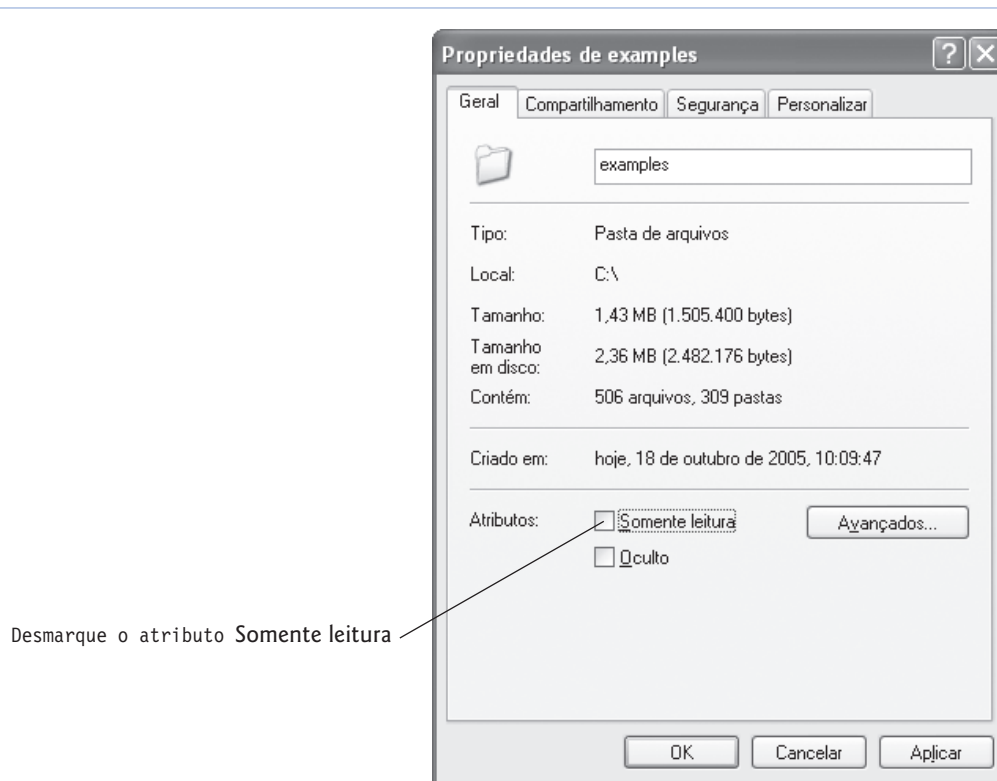


Figura 5 Desmarcando a caixa de seleção **Somente leitura**.

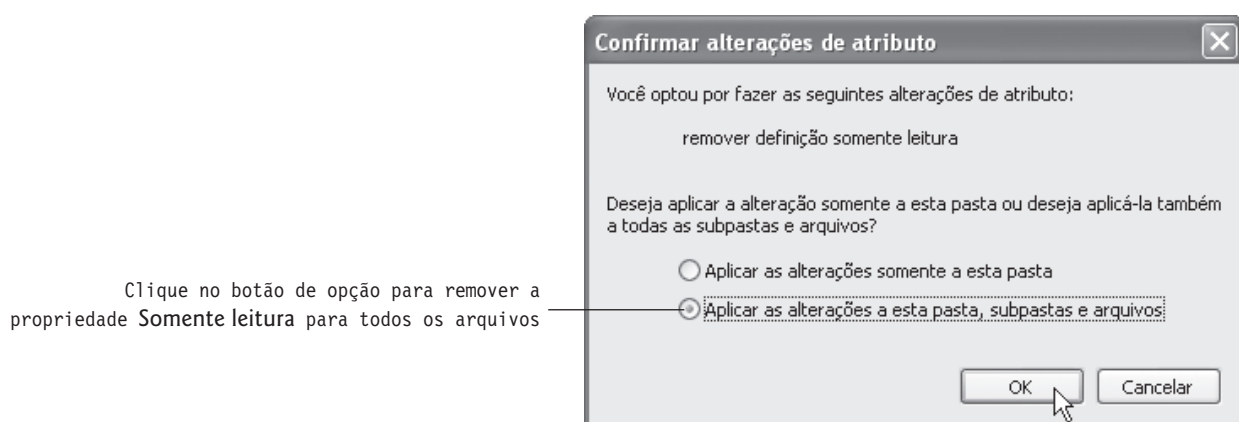


Figura 6 Removendo o atributo **Somente leitura** para todos os arquivos no diretório `examples`.



O principal mérito da língua é a clareza.
Galeno

Nossa vida é desperdiçada em detalhes... Simplifique, simplifique.
Henry David Thoreau

Ele tinha um talento maravilhoso para empacotar bem os pensamentos e torná-los portáteis.
Thomas B. Macaulay

O homem ainda é o computador mais extraordinário.
John F. Kennedy

Introdução aos computadores, à Internet e à World Wide Web

OBJETIVOS

Neste capítulo, você aprenderá:

- Conceitos básicos de hardware e software.
- Conceitos básicos de tecnologia de objeto, como classes, objetos, atributos, comportamentos, encapsulamento e herança.
- Os diferentes tipos de linguagens de programação.
- As linguagens de programação que são mais amplamente utilizadas.
- Um típico ambiente de desenvolvimento de programa em C++.
- A história da linguagem orientada a objetos usada como sistema de modelagem padrão da indústria, a UML.
- A história da Internet e da World Wide Web.
- A fazer um test-drive de aplicativos C++ em dois ambientes C++ populares — o GNU C++, que executa em Linux e o Visual C++®.NET da Microsoft, que executa no Windows® XP.

- 1.1 Introdução
- 1.2 O que é um computador?
- 1.3 Organização do computador
- 1.4 Primeiros sistemas operacionais
- 1.5 Computação pessoal distribuída e computação cliente/servidor
- 1.6 A Internet e a World Wide Web
- 1.7 Linguagens de máquina, linguagens assembly e linguagens de alto nível
- 1.8 História do C e do C++
- 1.9 C++ Standard Library
- 1.10 História do Java
- 1.11 Fortran, COBOL, Ada e Pascal
- 1.12 Basic, Visual Basic, Visual C++, C# e .NET
- 1.13 Tendência-chave do software: tecnologia de objeto
- 1.14 Ambiente de desenvolvimento C++ típico
- 1.15 Notas sobre o C++ e este livro
- 1.16 Test-drive de um aplicativo C++
- 1.17 Estudo de caso de engenharia de software: introdução à tecnologia de objetos e à UML (obrigatório)
- 1.18 Síntese
- 1.19 Recursos na Web

Resumo | Terminologia | Exercícios de revisão | Respostas dos exercícios de revisão | Exercícios

1.1 Introdução

Bem-vindo ao C++! Trabalhamos muito para criar o que esperamos que você julgue ser uma experiência de aprendizagem informativa, divertida e desafiante. O C++ é uma poderosa linguagem de programação de computador apropriada para pessoas tecnicamente orientadas com pouca ou nenhuma experiência em programação e para programadores experientes utilizarem na construção de substanciais sistemas de informações. *C++ Como programar, Quinta edição*, é uma ferramenta de aprendizagem eficaz para cada uma dessas audiências.

A essência do livro enfatiza como alcançar clareza de programa por meio de comprovadas técnicas de programação orientada a objetos — este é um livro de ‘classes e objetos desde o início’ — e os não-programadores aprenderão a programação de maneira certa desde o início. A apresentação é clara, direta e abundantemente ilustrada. Ensina os recursos do C++ no contexto de programas C++ funcionais completos e mostramos as saídas produzidas quando esses programas são executados em um computador — chamamos isso de **abordagem live-code (código ativo)**. Os programas de exemplo são incluídos no CD que acompanha este livro ou você pode fazer download desses programas em www.deitel.com ou www.prenhall.com/deitel.

Os primeiros capítulos introduzem os princípios básicos sobre computadores, programação de computadores e a linguagem de programação de computadores em C++, fornecendo uma base sólida para o tratamento mais profundo do C++ nos capítulos posteriores. Programadores experientes tendem a ler os primeiros capítulos rapidamente e, então, acham o tratamento de C++ no restante do livro tanto rigoroso como desafiador.

A maioria das pessoas está familiarizada de alguma maneira com as coisas empolgantes que os computadores fazem. Utilizando este livro-texto, você aprenderá a comandar computadores para fazer essas coisas. Os computadores (frequentemente referidos como **hardware**) são controlados por **software** (isto é, as instruções que você escreve para instruir o computador a realizar **ações** e tomar **decisões**). O C++ é uma das mais populares linguagens de desenvolvimento de software atualmente. Este texto fornece uma introdução à programação na versão de C++ padronizada nos Estados Unidos pelo **American National Standards Institute (ANSI)** e em todo o mundo pelos esforços da **International Organization for Standardization (ISO)**.

O uso de computadores está aumentando em quase todos os campos de trabalho. Os custos de computação têm diminuído radicalmente por causa do rápido desenvolvimento em tecnologia de hardware e software. Os computadores que ocupavam grandes salas e custavam milhões de dólares há algumas décadas agora podem ser gravados em chips de silício menores que uma unha, ao custo de apenas alguns poucos dólares. (Aqueles computadores grandes eram chamados de **mainframes** e são amplamente utilizados hoje em negócios, órgãos do governo e na indústria.) Felizmente, o silício é um dos materiais mais abundantes na terra — é um ingrediente da areia comum. A tecnologia do chip de silício tornou a computação tão econômica que um bilhão de computadores de uso geral estão em utilização em todo o mundo, auxiliando pessoas no comércio, indústria, governo e na vida pessoal.

Ao longo dos anos, muitos programadores aprenderam a metodologia de programação chamada programação estruturada. Você aprenderá a programação estruturada e uma estimulante e mais recente metodologia, a programação orientada a objetos. Por que ensinamos

ambas? Atualmente, a orientação a objeto é a metodologia de programação chave utilizada pelos programadores. Você criará e trabalhará com muitos objetos de software neste texto. Entretanto, descobrirá que sua estrutura interna é muitas vezes construída com técnicas de programação estruturada. Além disso, a lógica de manipular objetos é ocasionalmente expressa com programação estruturada.

Você está embarcando em um caminho desafiante e recompensador. À medida que avança, se você tiver qualquer dúvida, envie um e-mail para

deitel@deitel.com

Responderemos prontamente. Para manter-se atualizado com o desenvolvimento do C++ na Deitel & Associates, assine nosso boletim gratuito de correio eletrônico, *The Deitel Buzz Online*, em

www.deitel.com/newsletter/subscribe.html

Esperamos que você goste de aprender com *C++ Como programar, Quinta edição*.

1.2 O que é um computador?

Um **computador** é um dispositivo capaz de realizar computações e tomar decisões lógicas milhões e (até bilhões) de vezes mais rapidamente que o homem. Por exemplo, muitos computadores pessoais de hoje podem realizar um bilhão de adições por segundo. Uma pessoa operando uma calculadora de mesa pode gastar a vida toda para fazer cálculos e ainda não concluir a mesma quantidade de cálculos que um poderoso computador pessoal pode realizar em um segundo! (Questões a ponderar: Como você saberia se a pessoa somou os números corretamente? Como você saberia se o computador somou os números corretamente?) Os **supercomputadores** mais rápidos de hoje podem realizar trilhões de adições por segundo!

Os computadores processam **dados** sob o controle de conjuntos de instruções chamados **programas de computador**. Esses programas orientam o computador por meio de conjuntos ordenados de ações especificadas por pessoas chamadas **programadores de computador**.

Um computador consiste em vários dispositivos referidos como hardware (por exemplo, o teclado, a tela, o mouse, o disco rígido, a memória, os DVDs e as unidades de processamento). Os programas que executam em um computador são referidos como software. Os custos de hardware têm caído significativamente nos últimos anos, a ponto de os computadores pessoais terem se tornado um bem de consumo popular. Neste livro, você aprenderá métodos comprovados que estão reduzindo custos de desenvolvimento de software — a programação orientada a objetos e (em nosso Estudo de caso opcional de engenharia de software nos capítulos 2–7, 9 e 13) o projeto orientado a objetos.

1.3 Organização do computador

Independentemente das diferenças na aparência física, praticamente todos os computadores podem ser considerados como divididos em seis **unidades lógicas** ou seções.

1. **Unidade de entrada.** Esta é a seção ‘receptora’ do computador. Ele obtém informações (dados e programas de computador) de **dispositivos de entrada** e coloca essas informações à disposição das outras unidades para processamento. A maioria das informações é inserida em computadores por meio de dispositivos de entrada, como teclados e mouse. As informações também podem ser inseridas de muitas outras maneiras, incluindo falar com seu computador, digitalizar imagens e fazer seu computador receber informações de uma rede, como a **Internet**.
2. **Unidade de saída.** Esta é a seção de ‘envio’ do computador. Ela pega as informações que o computador processou e as coloca em vários **dispositivos de saída** para tornar as informações disponíveis à utilização fora do computador. A maioria das informações enviadas para a saída de computadores é exibida em telas, impressas em papel ou utilizadas para controlar outros dispositivos. Os computadores também podem gerar saída de suas informações para redes, como a Internet.
3. **Unidade de memória.** Esta é a seção de armazenamento de relativamente baixa capacidade e rápido acesso do computador. Ela armazena programas de computador enquanto estão sendo executados. Retém informações que foram inseridas pela unidade de entrada, para se tornarem imediatamente disponíveis para processamento quando necessário. A unidade de memória também retém informações processadas até que elas possam ser colocadas em dispositivos de saída pela unidade de saída. As informações na unidade de memória são, em geral, perdidas quando o computador é desligado. A unidade de memória costuma ser chamada de **memória** ou **memória principal**. [Historicamente, essa unidade era chamada de ‘memória básica’, mas o uso desse termo está desaparecendo gradualmente hoje.]
4. **Unidade de aritmética e lógica (*arithmetic and logic unit – ALU*).** Esta é a seção de ‘produção’ do computador. Ela é responsável pela realização de cálculos, como adição, subtração, multiplicação e divisão. Contém os mecanismos de decisão que permitem ao computador, por exemplo, comparar dois itens da unidade de memória para determinar se são iguais ou não.
5. **Unidade central de processamento (*central processing unit – CPU*).** Esta é a seção ‘administrativa’ do computador. Ela coordena e supervisiona a operação das outras seções. A CPU diz à unidade de entrada quando as informações devem ser lidas e transferidas para a unidade de memória, informa à ALU quando as informações da unidade de memória devem ser utilizadas em cálculos e instrui a unidade de saída sobre quando enviar as informações da unidade de memória para certos dispositivos de

saída. Muitos computadores de hoje têm múltiplas CPUs e, portanto, podem realizar muitas operações simultaneamente — esses computadores são chamados de **multiprocessadores**.

6. **Unidade de armazenamento secundária.** Esta é a seção de armazenamento de alta capacidade e longo prazo do computador. Programas ou dados que não são utilizados ativamente pelas outras unidades, em geral, são colocados em dispositivos de armazenamento secundário, como as unidades de disco, até que sejam novamente necessários, possivelmente horas, dias, meses ou mesmo anos mais tarde. As informações no armazenamento secundário exigem muito mais tempo para serem acessadas do que as informações na memória principal, mas o custo por unidade de armazenamento secundário é muito menor que o da memória principal. Outros dispositivos de armazenamento secundários incluem CDs e DVDs, que podem armazenar centenas de milhões de caracteres e bilhões de caracteres, respectivamente.

1.4 Primeiros sistemas operacionais

Os primeiros computadores podiam realizar apenas um **trabalho** ou **tarefa** por vez. Isso costuma ser chamado de **processamento em lotes** de um único usuário. O computador executa um único programa por vez enquanto está processando dados em grupos ou **lotes**. Nesses antigos sistemas, os usuários geralmente enviavam os trabalhos para um centro de processamento de dados em maços de cartões perfurados e, com frequência, tinham de esperar horas ou até mesmo dias antes de as impressões retornarem para suas mesas.

Os sistemas de software chamados **sistemas operacionais** foram desenvolvidos para tornar a utilização de computadores mais conveniente. Os primeiros sistemas operacionais tornaram a transição entre trabalhos mais suave e mais rápida e, portanto, aumentaram a quantidade de trabalho, ou **throughput**, que os computadores poderiam processar.

Quando os computadores se tornaram mais poderosos, ficou evidente que o processamento em lotes de um único usuário era ineficiente, porque uma grande quantidade de tempo era gasta esperando dispositivos de entrada/saída lentos completarem suas tarefas. Pensou-se que muitos trabalhos ou tarefas poderiam *compartilhar* os recursos do computador para alcançar melhor utilização. Isso é alcançado pela **multiprogramação**. A multiprogramação envolve a operação simultânea de muitos trabalhos que competem para compartilhar os recursos do computador. Com os primeiros sistemas operacionais baseados em multiprogramação, os usuários ainda submetiam trabalhos em unidades de cartões perfurados e esperavam horas ou dias para resultados.

Na década de 1960, vários grupos na indústria e nas universidades foram os pioneiros dos sistemas operacionais de **compartilhamento de tempo**. O compartilhamento de tempo é um caso especial da multiprogramação em que usuários acessam o computador por meio de terminais, em geral dispositivos com teclados e telas. Dúzias ou até mesmo centenas de usuários compartilham o computador de uma vez. Na verdade, o computador não executa todos eles simultaneamente. Em vez disso, executa uma pequena parte do trabalho de um usuário e, em seguida, atende o próximo usuário, talvez fornecendo serviço para cada usuário várias vezes por segundo. Portanto, os programas dos usuários parecem estar executando simultaneamente. Uma vantagem do compartilhamento de tempo é que as solicitações de usuários recebem respostas quase imediatas.

1.5 Computação pessoal distribuída e computação cliente/servidor

Em 1977, a Apple Computer popularizou a **computação pessoal**. Os computadores tornaram-se tão econômicos que as pessoas podiam comprá-los para utilização pessoal ou profissional. Em 1981, a IBM, o maior fornecedor de computadores do mundo, lançou o IBM Personal Computer. Essa computação pessoal rapidamente legitimada em negócios, indústria e órgãos do governo, como mainframes IBM, era intensamente utilizada.

Esses computadores eram unidades ‘independentes’ — as pessoas transportavam discos de um lado para o outro para o compartilhamento de informações (frequentemente chamado de ‘rede peão’). Embora computadores pessoais antigos não fossem suficientemente poderosos para compartilhamento de tempo de vários usuários, essas máquinas podiam ser interconectadas em redes de computadores, às vezes por linhas telefônicas e às vezes em **redes locais (local area networks – LANs)** dentro de uma organização. Isso levou ao fenômeno da **computação distribuída**, em que a computação de uma organização, em vez de ser realizada somente em alguma central de processamento, é distribuída por redes para os sites em que o trabalho da organização é realizado. Os computadores pessoais eram poderosos o bastante para tratar os requisitos de computação de usuários individuais, bem como as tarefas básicas de comunicação de passar eletronicamente informações entre computadores.

Os computadores pessoais de hoje são tão poderosos quanto as máquinas de milhões de dólares de apenas algumas décadas atrás. As máquinas desktop mais poderosas — chamadas **estações de trabalho** — fornecem enormes capacidades a usuários individuais. As informações são facilmente compartilhadas através de redes de computadores em que computadores chamados **servidores de arquivos** oferecem um armazenamento de dados comum que pode ser utilizado por computadores **clientes** distribuídos por toda a rede, daí o termo **computação cliente/servidor**. O C++ tornou-se largamente utilizado para escrever software para sistemas operacionais, redes de computadores e aplicativos cliente/servidor distribuídos. Sistemas operacionais populares de hoje, como UNIX, Linux, Mac OS X e sistemas baseados no Windows da Microsoft, fornecem os tipos de capacidades discutidas nesta seção.

1.6 A Internet e a World Wide Web

A **Internet** — uma rede global de computadores — foi iniciada há quase quatro décadas com o financiamento fornecido pelo Departamento de Defesa dos Estados Unidos. Originalmente projetada para conectar os principais sistemas de computador de cerca de uma dúzia de universidades e organizações de pesquisa, a Internet hoje é acessível por computadores em todo o mundo.

Com o surgimento da **World Wide Web** — que permite aos usuários de computador localizar e visualizar documentos baseados em multimídia sobre quase qualquer assunto pela Internet — a Internet explodiu, tornando-se um dos principais mecanismos de comunicação do mundo.

A Internet e a World Wide Web estão certamente entre as criações mais importantes e profundas da raça humana. Antigamente, a maioria dos aplicativos de computador era executada em computadores que não se comunicavam entre si. Os aplicativos de hoje podem ser escritos para se comunicar entre os computadores do mundo. A Internet funde computação e tecnologias de comunicação. Ela torna nosso trabalho mais fácil. Ela torna informações acessíveis mundialmente de maneira instantânea e conveniente. Permite aos indivíduos e às empresas de pequeno porte local obter exposição mundial. E está mudando a maneira como os negócios são feitos. As pessoas podem procurar os melhores preços em praticamente todos os produtos ou serviços. Comunidades de interesse especial podem permanecer em contato entre si. Os pesquisadores podem tornar-se instantaneamente cientes dos últimos avanços científicos. Depois de dominar o Capítulo 19, “Programação Web”, você será capaz de desenvolver aplicativos de computador baseados na Internet.

1.7 Linguagens de máquina, linguagens assembly e linguagens de alto nível

Os programadores escrevem instruções em várias linguagens de programação, algumas diretamente compreensíveis por computadores, e outras requerendo passos intermediários de **tradução**. Centenas de linguagens de computador estão atualmente em uso. Essas linguagens podem ser divididas em três tipos gerais:

1. Linguagens de máquina
2. Linguagens assembly
3. Linguagens de alto nível

Qualquer computador pode entender diretamente somente sua própria **linguagem de máquina**. A linguagem de máquina é a ‘linguagem natural’ de um computador e como tal é definida pelo seu design de hardware. [Nota: A linguagem de máquina é, muitas vezes, referida como **código-objeto**. Este termo é anterior à ‘programação orientada a objetos’. Essas duas utilizações de ‘objeto’ estão relacionadas.] As linguagens de máquina consistem geralmente em strings de números (em última instância reduzidas a 1s e 0s) que instruem os computadores a realizar suas operações mais elementares uma de cada vez. As linguagens de máquina são **dependentes de máquina** (isto é, uma linguagem particular de máquina pode ser utilizada apenas em um tipo de computador). Essas linguagens são muito complexas para os seres humanos, como ilustrado pela próxima seção de um antigo programa de linguagem de máquina que soma o ganho em horas extras ao salário-base e armazena o resultado no salário bruto:

```
+1300042774
+1400593419
+1200274027
```

A programação de linguagem de máquina era simplesmente muito lenta, tediosa e propensa a erro para a maioria dos programadores. Em vez de utilizar as strings de números que os computadores poderiam entender diretamente, os programadores começaram a utilizar abreviações em inglês para representar operações elementares. Essas abreviações formaram a base das **linguagens assembly**. **Programas tradutores** chamados **assemblers** foram desenvolvidos para converter os primeiros programas de linguagem assembly em linguagem de máquina a velocidades de computador. A seção a seguir de um programa de linguagem assembly também soma os ganhos em horas extras ao salário-base e armazena o resultado no salário bruto:

```
load      basepay
add       overpay
store     grosspay
```

Embora tal código seja mais claro para humanos, ele é incompreensível para computadores até ser traduzido em linguagem de máquina.

O uso de computadores aumentou rapidamente com o advento de linguagens assembly, mas os programadores ainda tinham de utilizar muitas instruções para realizar até as tarefas mais simples. Para acelerar o processo de programação foram desenvolvidas **linguagens de alto nível**, em que instruções únicas poderiam ser escritas para realizar tarefas substanciais. Os programas tradutores chamados **compiladores** convertem os programas de linguagem de alto nível em linguagem de máquina. Linguagens de alto nível permitem aos programadores escrever instruções que se parecem com o inglês cotidiano e contêm notações matemáticas comumente utilizadas. Um programa de folha de pagamentos escrito em uma linguagem de alto nível poderia conter uma instrução assim

```
grossPay = basePay + overTimePay;
```

Do ponto de vista do programador, obviamente, as linguagens de alto nível são preferíveis à linguagem de máquina e linguagem assembly. O C, o C++, as linguagens .NET da Microsoft (por exemplo, Visual Basic .NET, Visual C++ .NET e C#) e o Java estão entre as linguagens de programação de alto nível mais amplamente utilizadas.

O processo de compilação de um programa de linguagem de alto nível em linguagem de máquina pode consumir uma quantidade considerável de tempo de processamento. Programas **interpretadores** foram desenvolvidos para executar programas de linguagem de alto nível diretamente, embora muito mais lentamente. Os interpretadores são populares em ambientes de desenvolvimento de programa em que novos recursos são adicionados, e os erros, corrigidos. Depois que um programa está completamente desenvolvido, uma versão compilada pode ser produzida para executar mais eficientemente.

1.8 História do C e do C++

O C++ desenvolveu-se a partir do C, que se desenvolveu a partir de duas linguagens de programação anteriores, o BCPL e o B. O BCPL foi desenvolvido em 1967 por Martin Richards como uma linguagem para escrever software de sistemas operacionais e compiladores para sistemas operacionais. Ken Thompson modelou muitos recursos em sua linguagem B com base nas suas contrapartes em BCPL, e utilizou o B para criar versões anteriores do sistema operacional UNIX na Bell Laboratories em 1970.

A linguagem C foi desenvolvida da linguagem B por Dennis Ritchie na Bell Laboratories. O C utiliza muitos conceitos importantes de BCPL e B. No início, o C tornou-se amplamente conhecido como a linguagem de desenvolvimento do sistema operacional UNIX. Hoje, a maioria de sistemas operacionais é escrita em C e/ou C++. O C está agora disponível para a maioria dos computadores e é independente de hardware. Com design cuidadoso, é possível escrever programas em C que sejam **portáteis** para mais computadores.

A utilização disseminada de C com vários tipos de computadores (às vezes chamados **plataformas de hardware**) infelizmente levou a muitas variações. Isso era um problema sério para desenvolvedores de programa, que precisavam escrever programas portáteis que executassem em várias plataformas. Uma versão-padrão de C era necessária. O American National Standards Institute (ANSI) cooperou com a International Organization for Standardization (ISO) para padronizar o C em todo o mundo; o documento-padrão em conjunto foi publicado em 1990 e é referido como *ANSI/ISO 9899: 1990*.



Dica de portabilidade 1.1

Como o C é uma linguagem padronizada, amplamente disponível e independente de hardware, os aplicativos escritos em C podem ser frequentemente executados com pouca ou nenhuma modificação em um amplo espectro de sistemas de computador.

O C++, uma extensão do C, foi desenvolvido por Bjarne Stroustrup no início da década de 1980 na Bell Laboratories. O C++ fornece vários recursos que aprimoram a linguagem C, mas, sobretudo, fornece capacidades para a **programação orientada a objetos**, POO.

Uma revolução na comunidade de softwares está em progresso. Construir software de maneira rápida, correta e econômica permanece um objetivo difícil de alcançar e isso em uma época em que a demanda por novos e mais poderosos programas de software cresce vertiginosamente. **Objetos** são essencialmente **componentes** de software reutilizáveis que modelam itens do mundo real. Os desenvolvedores de software estão descobrindo que utilizar um projeto e uma abordagem de implementação modulares e orientados a objetos pode torná-los muito mais produtivos do que seriam com o uso de técnicas anteriores de programação. Programas orientados a objetos são mais fáceis de entender, corrigir e modificar.

1.9 C++ Standard Library

Os programas C++ consistem em partes chamadas **classes** e **funções**. Você pode programar cada parte que precisar para formar um programa C++. Mas a maioria dos programadores em C++ tira proveito das ricas coleções de classes e funções existentes na **C++ Standard Library**. Portanto, na realidade há duas partes para aprender o “mundo” C++. A primeira é aprender a própria linguagem C++; a segunda é aprender a utilizar as classes e funções na C++ Standard Library. Por todo o livro, discutimos muitas dessas classes e funções. O livro de P. J. Plauger, *The Standard C Library* (Upper Saddle River, NJ: Prentice Hall PTR, 1992), é uma leitura obrigatória para programadores que precisam de um entendimento profundo sobre as funções de biblioteca ANSI C incluídas no C++, como implementá-las e como utilizá-las para escrever código portátil. Em geral, as bibliotecas de classes padrão são disponibilizadas por fornecedores de compilador. Muitas bibliotecas de classes de uso especial são disponibilizadas por fornecedores de softwares independentes.



Observação de engenharia de software 1.1

*Utilize uma abordagem de ‘blocos de construção’ para criar programas. Evite reinventar a roda. Utilize partes existentes sempre que possível. Chamada de **reutilização de software**, essa prática é fundamental para a programação orientada a objetos.*



Observação de engenharia de software 1.2

Ao programar em C++, você normalmente utilizará os seguintes blocos de construção: classes e funções da C++ Standard Library, classes e funções que você e seus colegas criam, e classes e funções de várias bibliotecas independentes populares.

Incluimos muitas **Observações de engenharia de software** por todo o livro para explicar conceitos que afetam e melhoram a arquitetura total e a qualidade de sistemas de software. Destacamos também outros tipos de dicas, incluindo **Boas práticas de programação** (para ajudá-lo a escrever programas mais claros, mais compreensíveis, mais sustentáveis e mais fáceis de testar e **depurar** — ou remover erros de programação), **Erros de programação comuns** (problemas para examinar e evitar), **Dicas de desempenho** (técnicas para escrever programas que executam mais rapidamente e utilizam menos memória), **Dicas de portabilidade** (técnicas para ajudá-lo a escrever programas que podem ser executados, com pouca ou nenhuma modificação, em uma variedade de computadores — essas dicas também incluem observações gerais sobre como o C++ alcança um alto grau de portabilidade) e **Dicas de prevenção de erros** (técnicas para remover bugs de programas e, sobretudo, técnicas para escrever programas livres de bugs em primeiro lugar). Muitas dessas técnicas são apenas diretrizes. Sem dúvida, você desenvolverá seu próprio estilo de programação preferido.

A vantagem de criar suas próprias funções e classes é que você saberá exatamente como elas funcionam. Você será capaz de examinar o código C++. A desvantagem é o demorado e complexo esforço para projetar, desenvolver e manter novas funções e classes que sejam corretas e operem eficientemente.



Dica de desempenho 1.1

Utilizar funções e classes da C++ Standard Library em vez de escrever suas próprias versões pode melhorar o desempenho do programa, porque elas são escritas cuidadosamente para executar com eficiência. Essa técnica também diminui o tempo de desenvolvimento de programa.



Dica de portabilidade 1.2

Utilizar funções e classes da C++ Standard Library em vez de escrever suas próprias funções e classes melhora a portabilidade do programa, porque elas são incluídas em cada implementação C++.

1.10 História do Java

Os microprocessadores têm um impacto profundo em dispositivos eletrônicos voltados para o consumo popular. Reconhecendo isso, a Sun Microsystems financiou, em 1991, um projeto de pesquisa corporativa interna com o codinome Green. O projeto resultou no desenvolvimento de uma linguagem baseada em C++ que seu criador, James Gosling, chamou de Oak em homenagem a uma árvore de carvalho vista por sua janela na Sun. Descobriu-se mais tarde que já havia uma linguagem de computador chamada Oak. Quando um grupo da Sun visitou uma cafeteria local, o nome **Java** (cidade de origem de um tipo de café importado) foi sugerido; e o nome pegou.

O projeto Green passou por algumas dificuldades. O mercado de dispositivos eletrônicos inteligentes voltados para o consumo popular não se desenvolvia no início da década de 1990 tão rapidamente quanto a Sun antecipara. O projeto corria o risco de ser cancelado. Por pura sorte, a World Wide Web explodiu em popularidade em 1993, e a Sun percebeu o imediato potencial de utilizar Java para adicionar **conteúdo dinâmico** (por exemplo, interatividade, animações e assim por diante) a páginas Web. Isso deu nova vida ao projeto.

Em 1995, a Sun anunciou formalmente o Java. O Java gerou interesse imediato na comunidade de negócios por causa do fenomenal sucesso da World Wide Web. O Java é agora utilizado para desenvolver aplicativos corporativos de grande porte, aprimorar a funcionalidade de servidores Web (os computadores que fornecem o conteúdo que vemos em nossos navegadores Web), fornecer aplicativos para dispositivos voltados para o consumo popular (como telefones celulares, pagers e PDAs) e para muitos outros propósitos. Versões atuais do C++, como o Visual C++® .NET da Microsoft® e o C++Builder™ da Borland®, têm capacidades semelhantes.

1.11 Fortran, COBOL, Ada e Pascal

Centenas de linguagens de alto nível foram desenvolvidas, mas somente algumas alcançaram ampla aceitação. O **FORTAN** (FORmula TRANslator) foi uma linguagem desenvolvida pela IBM Corporation em meados da década de 1950 para ser utilizada em aplicativos científicos de engenharia que exigem complexos cálculos matemáticos. O FORTRAN ainda é amplamente utilizado, especialmente em aplicativos de engenharia.

O **COBOL** (Common Business Oriented Language) foi desenvolvido no final da década de 1950 por fabricantes de computadores e usuários de computadores no governo norte-americano e na indústria. O COBOL é utilizado para aplicativos comerciais que exigem manipulação precisa e eficiente de grandes quantidades de dados. Grande parte do software utilizado em grandes empresas varejistas ainda é programada em COBOL.

Durante a década de 1960, muitos grandes esforços no desenvolvimento de software encontraram diversas dificuldades. Em geral, os prazos de entrega de software atrasavam, os custos excediam enormemente os orçamentos e os produtos finais não eram confiáveis. As pessoas começaram a perceber que o desenvolvimento de software era uma atividade muito mais complexa do que haviam imaginado. Uma pesquisa feita na década de 1960 resultou na evolução da **programação estruturada** — uma abordagem disciplinada para escrever programas mais claros, mais fáceis de testar e depurar e mais fáceis de modificar do que os grandes programas produzidos com a técnica anterior.

Um dos resultados mais tangíveis dessa pesquisa foi o desenvolvimento da linguagem de programação **Pascal** pelo professor Niklaus Wirth em 1971. Batizada em homenagem a Blaise Pascal, matemático e filósofo do século XVII, essa linguagem foi adotada para ensino de programação estruturada e rapidamente se tornou a linguagem de programação preferida na maioria das universidades. O Pascal não possui muitos recursos necessários em aplicativos comerciais, industriais e governamentais, assim não foi amplamente aceito nesses ambientes.

A linguagem de programação **Ada** foi desenvolvida sob o patrocínio do Departamento de Defesa (DOD) dos Estados Unidos durante a década de 1970 e o início da década de 1980. Centenas de linguagens separadas foram utilizadas a fim de produzir os sistemas de software para controle e comando maciços do DOD. O DOD queria que uma única linguagem atendesse a maioria de suas necessidades. A linguagem foi batizada como Ada em homenagem a Lady Ada Lovelace, filha do poeta Lord Byron. Lady Lovelace é considerada a primeira pessoa a escrever um programa de computador do mundo no início do século XIX (para o dispositivo mecânico de computação conhecido como Máquina Analítica, projetado por Charles Babbage). Uma importante capacidade da linguagem Ada,

chamada **multitarefa**, permite que os programadores especifiquem quantas atividades devem ocorrer paralelamente. O Java, por meio de uma técnica chamada **multithreading**, também permite que os programadores escrevam programas com atividades paralelas. Embora não faça parte do padrão C++, o **multithreading** é disponibilizado por várias bibliotecas de classes suplementares.

1.12 Basic, Visual Basic, Visual C++, C# e .NET

A linguagem de programação **BASIC** (Beginner's All-Purpose Symbolic Instruction Code) foi desenvolvida em meados da década de 1960, no Dartmouth College, como um meio de escrever programas simples. O principal propósito do BASIC foi familiarizar os iniciantes com as técnicas de programação. A linguagem Visual Basic da Microsoft, introduzida no início de 1990 para simplificar o desenvolvimento de aplicativos Microsoft Windows, tornou-se uma das linguagens de programação mais populares no mundo.

As últimas ferramentas de desenvolvimento da Microsoft fazem parte de sua estratégia de escopo corporativo para integrar a Internet e a Web em aplicativos de computador. Essa estratégia é implementada na **plataforma .NET** da Microsoft, que fornece aos desenvolvedores as capacidades necessárias para criar e executar aplicativos de computador que possam executar em computadores distribuídos através da Internet. As três principais linguagens de programação da Microsoft são **Visual Basic .NET** (baseada no BASIC original), **Visual C++ .NET** (baseada no C++) e **C#** (uma nova linguagem baseada no C++ e no Java que foi desenvolvida especificamente para a plataforma .NET). Os desenvolvedores que utilizam .NET podem escrever componentes de software na linguagem com os quais estão mais familiarizados e então formar aplicativos combinando esses componentes com componentes escritos em qualquer linguagem do .NET.

1.13 Tendência-chave do software: tecnologia de objeto

Um dos autores, Harvey Deitel, lembra-se da grande frustração que foi sentida, na década de 1960, por organizações de desenvolvimento de softwares, especialmente as que trabalhavam em projetos de larga escala. Durante seus anos universitários, ele teve o privilégio de trabalhar nas férias para um importante fornecedor de computador em equipes de desenvolvimento de sistemas operacionais de memória virtual e compartilhamento de tempo. Essa foi uma grande experiência para um aluno de faculdade. Mas, no verão de 1967, a realidade se impôs quando a empresa se “descomprometeu” a produzir comercialmente um sistema particular em que centenas das pessoas tinham trabalhado por muitos anos. Era difícil fazer esse software funcionar. Software é um “material complexo”.

Os aprimoramentos na tecnologia de software emergiram com os benefícios da programação estruturada (e as disciplinas relacionadas de **análise e design de sistemas estruturados**) sendo realizada na década de 1970. Mas foi somente quando a tecnologia de programação orientada a objetos se tornou amplamente utilizada na década de 1990 que os desenvolvedores de software sentiram, por fim, que tinham as ferramentas necessárias para dar passos importantes no processo de desenvolvimento de software.

De fato, a tecnologia de objetos data de meados de 1960. A linguagem de programação C++, desenvolvida na AT&T por Bjarne Stroustrup no início da década de 1980, é baseada em duas linguagens — o C, que foi inicialmente desenvolvido na AT&T para implementar o sistema operacional UNIX no início da década de 1970, e o Simula 67, uma linguagem de programação de simulação desenvolvida na Europa e lançada em 1967. O C++ absorveu os recursos do C e acrescentou capacidades do Simula para criar e manipular objetos. Nem o C nem o C++ foram originalmente projetados para larga utilização fora dos laboratórios de pesquisa da AT&T. Mas o suporte em grande escala se desenvolveu rapidamente para cada linguagem.

O que são objetos e por que eles são especiais? Na realidade, a tecnologia de objetos é um esquema de empacotamento que nos ajuda a criar unidades significativas de software. Estas podem ser grandes e fortemente focalizadas em áreas particulares de aplicativos. Há objetos data, objetos tempo, objetos cheque, objetos pagamento, objetos fatura, objetos áudio, objetos vídeo, objetos arquivo, objetos registro e assim por diante. De fato, quase todo substantivo pode ser razoavelmente representado como um objeto.

Vivemos em um mundo de objetos. Dê uma olhada à sua volta. Há carros, aviões, pessoas, animais, edifícios, sinais de trânsito, elevadores e muito mais. Antes de as linguagens orientadas a objetos aparecerem, as linguagens de programação (como FORTRAN, COBOL, Pascal, Basic e C) eram o foco das ações (verbos) em vez do foco de coisas ou objetos (substantivos). Os programadores vivendo em um mundo de objetos programavam utilizando principalmente verbos. Isso tornava a escrita de programas inconveniente. Agora, com a disponibilidade de linguagens orientadas a objetos populares, como C++ e Java, os programadores continuam a viver em um mundo orientado a objetos e podem programar de maneira orientada a objetos. Esse é um processo mais natural do que a programação procedural e resultou em aprimoramentos de produtividade significativos.

Um problema-chave com a programação procedural é que as unidades de programa não espelham facilmente as entidades do mundo real de modo eficaz, então essas unidades não são particularmente reutilizáveis. Não é incomum programadores recomencarem desde o início cada novo projeto e ter de escrever software semelhante “a partir do zero”. Isso desperdiça tempo e dinheiro, já que as pessoas “reinventam a roda” repetidamente. Com a tecnologia de objeto, as entidades de software criadas (chamadas **classes**), se adequadamente projetadas, tendem a ser muito mais reutilizáveis em projetos futuros. Utilizar bibliotecas de componentes reutilizáveis, como a **MFC (Microsoft Foundation Classes)**, a **.NET Framework Class Library da Microsoft** e aquelas produzidas pela Rogue Wave e por muitas outras organizações de desenvolvimento de software, pode reduzir significativamente a quantidade de esforço requerido para implementar certos tipos de sistemas (comparado ao esforço que seria necessário para reinventar essas capacidades em novos projetos).



Observação de engenharia de software 1.3

Extensas bibliotecas de classes de componentes de software reutilizáveis estão disponíveis pela Internet e World Wide Web. Muitas dessas bibliotecas estão disponíveis sem nenhuma taxa.

Algumas organizações afirmam que o benefício-chave que a programação orientada a objetos fornece não é a reutilização de software. Em vez disso, essas organizações argumentam que a POO tende a produzir software mais compreensível, mais organizado e mais fácil de manter, modificar e depurar. Isso pode ser significativo, porque foi estimado que até 80 por cento dos custos de software não estão associados aos esforços originais para o desenvolvimento do software, mas com a contínua evolução e manutenção desse software por todo o seu tempo de vida.

Quaisquer que sejam os benefícios percebidos da orientação a objeto, está claro que ela será a principal metodologia de programação pelas próximas décadas.

1.14 Ambiente de desenvolvimento C++ típico

Vamos considerar os passos na criação e execução de um aplicativo C++ utilizando um ambiente de desenvolvimento C++ (ilustrado na Figura 1.1). Em geral, os sistemas C++ consistem em três partes: um ambiente de desenvolvimento de programa, a linguagem e a C++ Standard Library. Os programas C++, em geral, passam por seis fases: **edição**, **pré-processamento**, **compilação**, **linkagem**, **carregamento** e **execução**. A discussão a seguir explica um típico ambiente de desenvolvimento de programa C++.

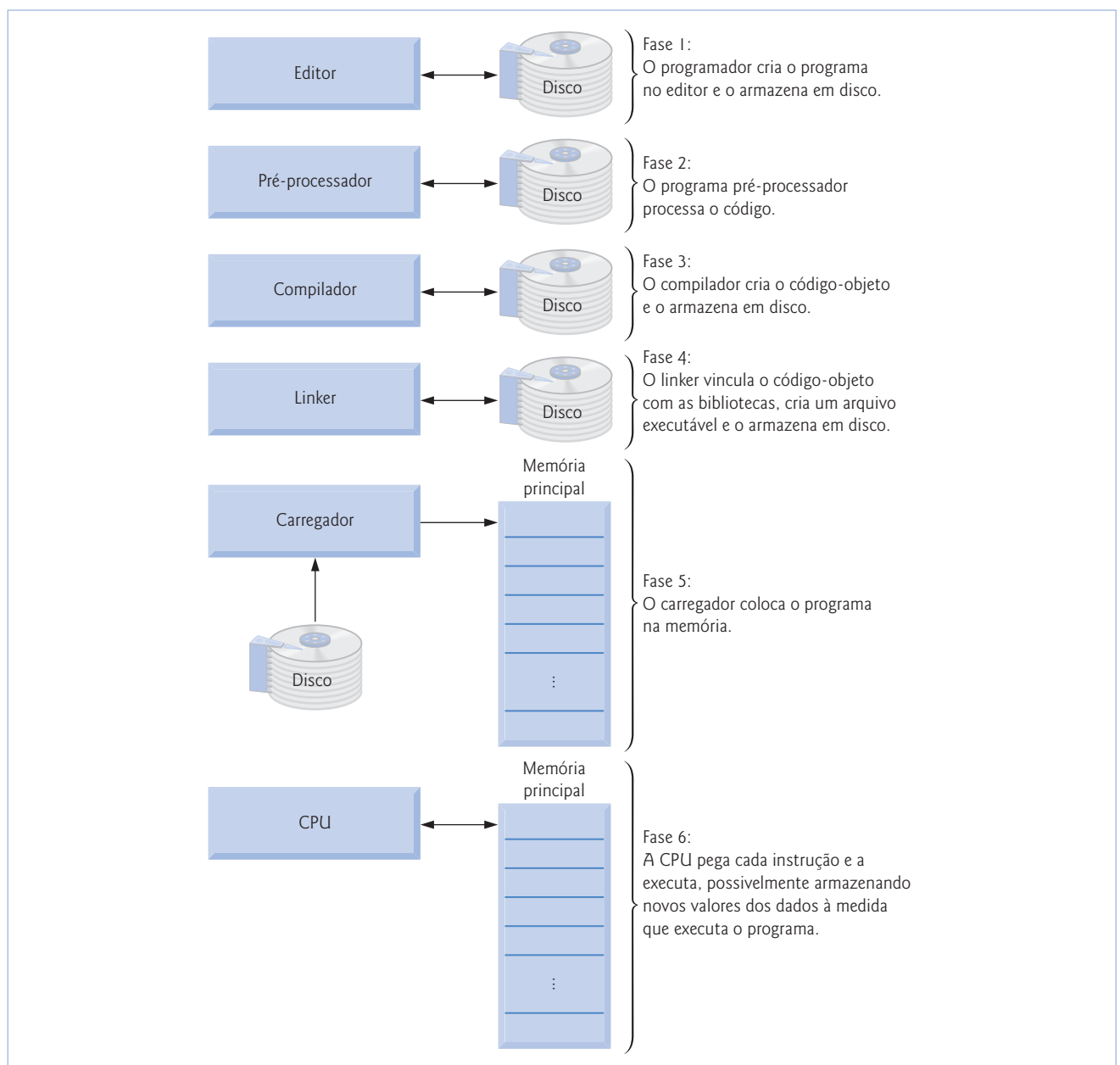


Figura 1.1 Típico ambiente C++.

em www.deitel.com/books/downloads.html, fornecemos as publicações da série Dive Into™ para ajudá-lo a começar a utilizar as várias ferramentas de desenvolvimento C++ populares, incluindo o Borland® C++Builder™ e o Microsoft® Visual C++® 6, o Microsoft® Visual C++® .NET, o GNU C++ em Linux e o GNU C++ no ambiente Cygwin™ UNIX® para Windows®. Tornaremos outras publicações da série Dive Into™ disponíveis à medida que instrutores as solicitarem.]

Fase 1: Criando um programa

A Fase 1 consiste em editar um arquivo com um **programa editor** (em geral, conhecido simplesmente como **editor**). Você digita um programa C++ (em geral referido como **código-fonte**) utilizando o editor, faz quaisquer correções necessárias e salva o programa em um dispositivo de armazenamento secundário, tal como sua unidade de disco. Os nomes de arquivo de código-fonte C++ costumam terminar com as extensões .cpp, .cxx, .cc ou .C (observe que C está em letra maiúscula), que indicam que um arquivo contém código-fonte C++. Consulte a documentação do seu ambiente C++ para obter informações adicionais sobre extensões de nome do arquivo.

Dois editores amplamente utilizados em sistemas UNIX são o vi e o emacs. Os pacotes de software C++ para Microsoft Windows como o Borland C++ (www.borland.com), o Metrowerks CodeWarrior (www.metrowerks.com) e o Microsoft Visual C++ (www.msdn.microsoft.com/visualc/) têm editores integrados no ambiente de programação. Você também pode utilizar um editor de textos simples, como o Bloco de notas (Notepad) no Windows, para escrever o código C++. Supomos que o leitor sabe como editar um arquivo.

Fases 2 e 3: Pré-processamento e compilação de um programa C++

Na Fase 2, o programador dá o comando para **compilar** o programa. Em um sistema C++, um programa **pré-processador** executa automaticamente antes de a fase de conversão do compilador iniciar (então chamamos a fase 2 de pré-processamento, e a fase 3, de compilação). O pré-processador C++ obedece a comandos chamados de **diretivas de pré-processador**, que indicam que certas manipulações devem ser realizadas no programa antes da compilação. Essas manipulações normalmente incluem outros arquivos de texto a ser compilados e realizam várias substituições de texto. As diretivas de pré-processador mais comuns são discutidas nos primeiros capítulos; uma discussão detalhada de todos os recursos de pré-processador aparece no Apêndice F, “Pré-processador”. Na fase 3, o compilador converte o programa C++ em código de linguagem de máquina (também referido como código-objeto).

Fase 4: Vinculando

A Fase 4 é chamada **linkagem**. Em geral, os programas C++ contêm referências a funções e dados definidos em outra parte, como nas bibliotecas-padrão ou nas bibliotecas privadas de grupos de programadores que trabalham em um projeto particular. Normalmente, o código-objeto produzido pelo compilador C++ contém ‘lacunas’ devido a essas partes ausentes. Um **linker** vincula o código-objeto com o código das funções ausentes para produzir uma **imagem executável** (sem partes ausentes). Se o programa compilar e vincular corretamente, uma imagem executável é produzida.

Fase 5: Carregamento

A Fase 5 é chamada **carregamento**. Antes que um programa possa ser executado, ele deve primeiro ser colocado na memória. Isso é feito pelo **carregador**, que pega a imagem executável do disco e a transfere para a memória. Componentes adicionais de bibliotecas compartilhadas que suportam o programa também são carregados.

Fase 6: Execução

Por fim, o computador, sob o controle da CPU, **executa** o programa uma instrução por vez.

Problemas que podem ocorrer em tempo de execução

Os programas nem sempre funcionam na primeira tentativa. Cada uma das fases anteriores pode falhar por causa de vários erros que discutimos por todo o livro. Por exemplo, um programa executável poderia tentar realizar uma operação de divisão por zero (uma operação ilegal para a aritmética de número inteiro em C++). Isso faria o programa C++ imprimir uma mensagem de erro. Se isso ocorresse, você teria de retornar à fase de edição, fazer as correções necessárias e passar novamente pelas demais fases para determinar se as correções resolveram o(s) problema(s).

A maioria dos programas em C++ realiza entrada e/ou saída de dados. Certas funções C++ obtêm sua entrada de **cin** (o **fluxo de entrada padrão**; pronuncia-se ‘ci-in’), que normalmente é o teclado, mas cin pode ser redirecionado para outro dispositivo. A saída dos dados é freqüentemente para **cout** (o **fluxo de saída padrão**; pronuncia-se ‘ci-out’), que, normalmente, é a tela do computador, mas cout pode ser redirecionado para outro dispositivo. Quando dizemos que um programa imprime um resultado, normalmente isso significa que o resultado é exibido em uma tela. Os dados podem ser enviados para outros dispositivos de saída como discos e impressoras. Também há um **fluxo de erro padrão** referido como **cerr**. O fluxo cerr (normalmente conectado à tela) é utilizado para exibir mensagens de erro. É comum usuários atribuírem cout a um dispositivo em vez de à tela enquanto mantêm cerr atribuído à tela, assim as saídas normais são separadas dos erros.



Erro comum de programação 1.1

Os erros, como divisão por zero, ocorrem enquanto um programa executa, então são chamados **erros de tempo de execução** ou **erros de runtime**. **Erros de tempo de execução fatais** fazem com que os programas sejam imediatamente encerrados sem terem realizado seus trabalhos com sucesso. **Erros de tempo de execução não fatais** permitem que os programas executem até sua conclusão, produ-

zindo freqüentemente resultados incorretos. [Nota: Em alguns sistemas, a divisão por zero não é um erro fatal. Veja a documentação do seu sistema.]

1.15 Notas sobre o C++ e este livro

Programadores em C++ experientes às vezes ficam orgulhosos por serem capazes de fazer algum uso exótico, desvirtuado ou distorcido da linguagem. Essa é uma prática de programação pobre. Ela torna os programas mais difíceis de ler, mais propensos a se comportar estranhamente, mais difíceis de testar e depurar e mais difíceis de adaptar em caso de alteração de requisitos. Este livro é projetado para programadores iniciantes, então salientamos a clareza do programa. Segue nossa primeira ‘boa prática de programação’.



Boa prática de programação 1.1

Escreva seus programas C++ de uma maneira simples e direta. Isso é às vezes chamado KIS (‘keep it simple’, ‘mantenha a coisa simples’). Não ‘estenda’ a linguagem tentando usos bizarros.

Você já ouviu que o C e o C++ são linguagens portáteis e que programas escritos em C e C++ podem ser executados em muitos computadores diferentes. Portabilidade é um objetivo vago. O documento-padrão C ANSI contém uma longa lista de questões de portabilidade; e livros completos foram escritos para discutir portabilidade.



Dica de portabilidade 1.3

*Embora seja possível escrever programas portáteis, há muitos problemas entre diferentes compiladores C e C++ e diferentes computadores que podem dificultar o alcance da portabilidade. Escrever programas em C e C++ não garante a portabilidade. Com freqüência, o programador precisará lidar diretamente com variações de compilador e computador. Como um grupo, esses problemas são às vezes chamados de variações de *plataforma*.*

Auditamos nossa apresentação com o documento ANSI/ISO C++ padrão para alcançar completude e exatidão. Entretanto, o C++ é uma linguagem rica e há alguns recursos que não abrangemos. Se precisar de detalhes técnicos adicionais sobre o C++, recomendamos que você leia o documento-padrão C++, que pode ser encomendado no site Web da ANSI em

webstore.ansi.org/ansidocstore/default.asp

O título do documento é ‘Information Technology – Programming Languages – C++’ e seu número de documento é INCITS/ISO/IEC 14882-2003.

Incluimos uma extensa bibliografia de livros e artigos sobre C++ e programação orientada a objetos. Além disso, incluímos um apêndice de recursos sobre C++ na Internet e em sites Web que se relacionam com C++ e a programação orientada a objetos. Listamos vários sites Web na Seção 1.19, incluindo links para compiladores C++ gratuitos, sites de recursos e alguns jogos C++ divertidos e tutoriais de programação de jogos.



Boa prática de programação 1.2

Leia os manuais da versão de C++ que você está utilizando. Consulte com freqüência esses manuais para certificar-se de que está ciente da rica coleção de recursos C++ e de que os está utilizando corretamente.



Boa prática de programação 1.3

Seu computador e seu compilador são bons professores. Se depois de ler o manual da linguagem C++ você ainda não estiver seguro da maneira como um recurso C++ funciona, experimente utilizar um pequeno ‘programa de teste’ e ver o que acontece. Configure suas opções de compilador para ‘avisos máximos’. Estude cada mensagem gerada pelo compilador e corrija os programas para eliminar as mensagens.

1.16 Test-drive de um aplicativo C++

Nesta seção, você irá executar e interagir com seu primeiro aplicativo C++. Você começará executando um interessante jogo de adivinhação de números, que escolhe um número de 1 a 1.000 e pede para você adivinhar o número. Se sua suposição estiver correta, o jogo termina. Se não estiver correta, o aplicativo indica se sua suposição é um número maior ou menor que o correto. Não há limite quanto ao número de suposições que você pode fazer. [Nota: Apenas para este test-drive, modificamos esse aplicativo a partir do exercício que você será solicitado a criar no Capítulo 6, “Funções e uma introdução à recursão”. Em geral, esse aplicativo seleciona números diferentes para você adivinhar toda vez que executá-lo, porque ele escolhe os números aleatoriamente. Nosso aplicativo modificado escolhe as mesmas suposições ‘corretas’ toda vez que você executar o programa. Isso permite utilizar as mesmas suposições e ver os mesmos resultados que mostramos quando orientamos você pela interação com seu primeiro aplicativo C++.]

Demonstraremos como executar um aplicativo C++ de duas maneiras — utilizando o Prompt do MS-DOS do Windows XP e utilizando um shell no Linux (semelhante a um Prompt do MS-DOS do Windows). O aplicativo executa de maneira semelhante em ambas as

plataformas. Muitos ambientes de desenvolvimento que estão disponíveis podem compilar, construir e executar aplicativos C++, como C++Builder da Borland, Metrowerks, GNU C++, Visual C++ .Net da Microsoft etc. Enquanto não fazemos test-drive de cada um desses ambientes, fornecemos informações na Seção 1.19 relacionadas com compiladores C++ disponíveis para download na Internet. Consulte seu instrutor para obter informações sobre seu ambiente de desenvolvimento específico. Além disso, fornecemos várias publicações da série *Dive Into™* para ajudá-lo na introdução com os vários compiladores C++. Esses estão gratuitamente disponíveis para download em www.deitel.com/books/cpphttp5/index.html.

Nos passos seguintes, você executará o aplicativo e inserirá vários números para adivinhar o número correto. Os elementos e a funcionalidade que você vê nesse aplicativo são típicos daqueles que você aprenderá a programar neste livro. Por todo o livro, utilizamos fontes para distinguir entre os recursos que você vê na tela (por exemplo, o Prompt do MS-DOS) e elementos que não são diretamente relacionados com a tela. Nossa convenção serve para enfatizar elementos de tela como títulos e menus (por exemplo, o menu Arquivo) em uma fonte Gouldy sans e enfatizar nomes de arquivo, texto exibido por um aplicativo e valores que você deve inserir em um aplicativo (por exemplo, GuessNumber ou 500), em uma fonte Letter Gothic. Como você já notou, a **ocorrência da definição** de cada termo aparece em azul e negrito. Para as figuras nesta seção, destacamos a entrada de usuário necessária a cada passo e indicamos as partes significativas do aplicativo. Para tornar esses recursos mais visíveis, modificamos a cor de fundo da janela Prompt do MS-DOS (apenas para o test-drive do Windows). Para modificar as cores do Prompt do MS-DOS no sistema, abra um Prompt do MS-DOS, dê um clique com botão direito do mouse na barra de título e selecione Propriedades. Na caixa de diálogo Propriedades de 'C:\WINDOWS\system32\ntvdm.exe' que aparece, clique na guia Cores e selecione o texto e as cores de fundo preferidas.

Executando um aplicativo C++ a partir do Prompt do MS-DOS do Windows XP

1. **Verificando sua configuração.** Leia a seção *Antes de começar* deste livro-texto para se certificar de que você copiou corretamente os exemplos do livro para a unidade de disco.
2. **Localizando o aplicativo concluído.** Abra uma janela Prompt de comando. Para os leitores que utilizam o Windows 95, 98 ou 2000, selecione Iniciar > Programas > Acessórios > Prompt do MS-DOS. Para os usuários do Windows XP, selecione Iniciar > Todos os programas > Acessórios > Prompt de comando. Para mudar para o diretório de aplicativo GuessNumber completado, digite **cd C:\examples\ch01\GuessNumber\Windows**, depois pressione *Enter* (Figura 1.2). O comando **cd** é utilizado para alterar diretórios.
3. **Executando o aplicativo GuessNumber.** Agora que você está no diretório que contém o aplicativo GuessNumber, digite o comando **GuessNumber** (Figura 1.3) e pressione *Enter*. [Nota: GuessNumber.exe é o nome real do aplicativo; entretanto, o Windows assume a extensão .exe por padrão.]
4. **Inserindo sua primeira suposição.** O aplicativo exibe "Please type your first guess.", então exibe um ponto de interrogação (?) como um prompt na próxima linha (Figura 1.3). No prompt, insira **500** (Figura 1.4).
5. **Inserindo outra suposição.** O aplicativo exibe "Too high. Try again.", o que significa que o valor que você inseriu é maior do que o escolhido pelo aplicativo como a suposição correta. Portanto, você deve inserir um número menor na próxima suposição. No prompt, insira **250** (Figura 1.5). O aplicativo exibe novamente "Too high. Try again.", porque o valor inserido ainda é maior do que o número da suposição correta.

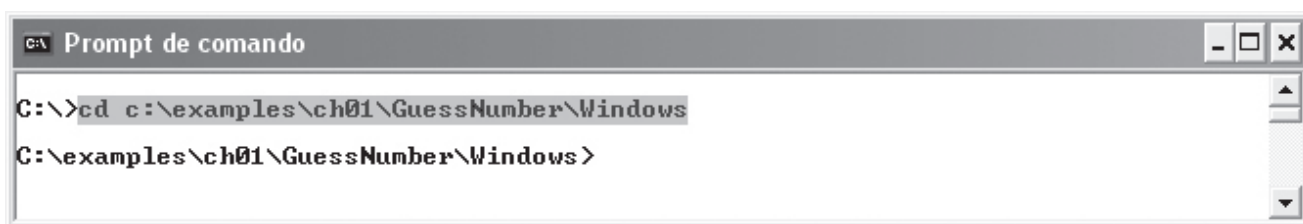
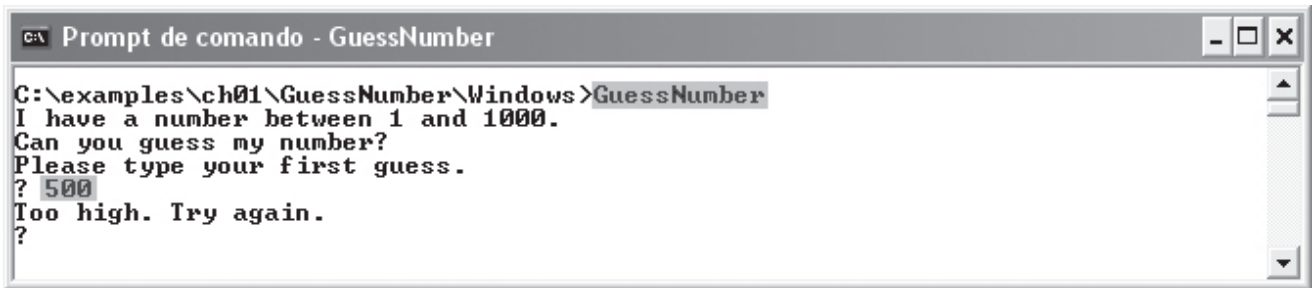


Figura 1.2 Abrindo uma janela Prompt do MS-DOS e alterando o diretório.

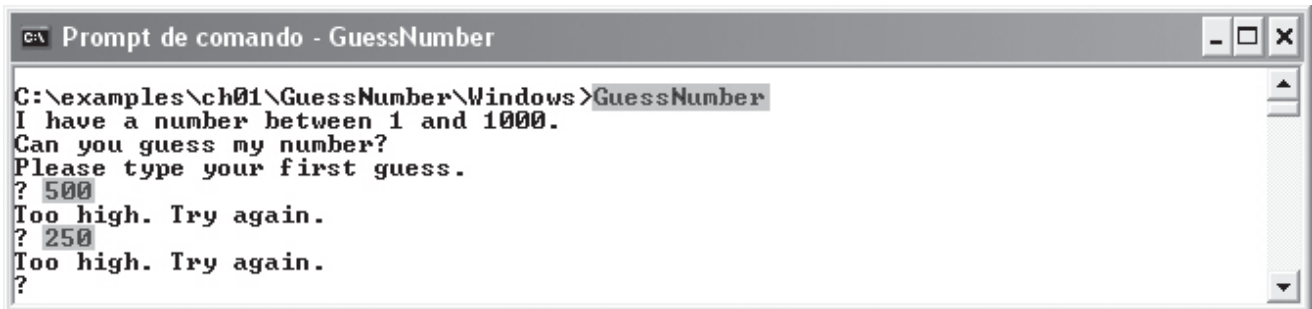


Figura 1.3 Executando o aplicativo GuessNumber.



```
C:\examples\ch01\GuessNumber\Windows>GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
?
```

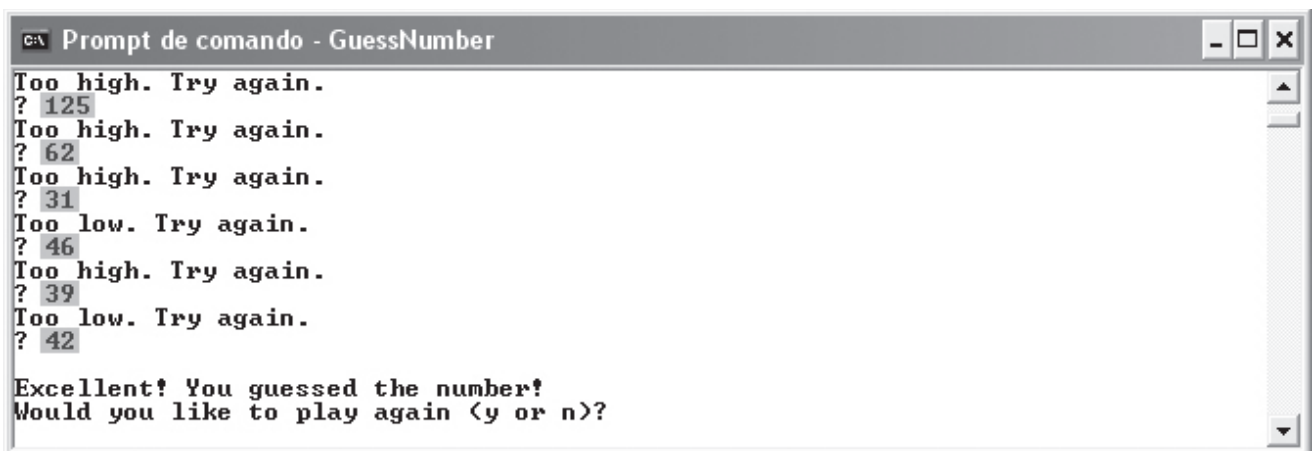
Figura 1.4 Inserindo sua primeira suposição.



```
C:\examples\ch01\GuessNumber\Windows>GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
? 250
Too high. Try again.
?
```

Figura 1.5 Inserindo uma segunda suposição e recebendo feedback.

6. *Inserindo suposições adicionais.* Continue o jogo inserindo valores até adivinhar o número correto. Uma vez que adivinhar a resposta, o aplicativo exibirá "Excellent! You guessed the number!" (Figura 1.6).
7. *Jogando novamente ou saindo do aplicativo.* Depois de adivinhar o número correto, o aplicativo pergunta se você gostaria de ir para outro jogo (Figura 1.6). No prompt "Would you like to play again (y or n)?", inserir o caractere **y** faz com que o aplicativo escolha um novo número e exiba a mensagem "Please enter your first guess." seguida por um prompt de ponto de interrogação (Figura 1.7) para que você possa fazer sua primeira suposição no novo jogo. Inserir o caractere **n** fecha o aplicativo e o retorna para o diretório do aplicativo no Prompt do MS-DOS (Figura 1.8). Toda vez que você executar esse aplicativo desde o início (isto é, *Passo 3*), ele escolherá os mesmos números para você adivinhar.
8. *Feche a janela Prompt do MS-DOS.*



```
C:\examples\ch01\GuessNumber\Windows>GuessNumber
Too high. Try again.
? 125
Too high. Try again.
? 62
Too high. Try again.
? 31
Too low. Try again.
? 46
Too high. Try again.
? 39
Too low. Try again.
? 42
Excellent! You guessed the number!
Would you like to play again (y or n)?
```

Figura 1.6 Inserindo suposições adicionais e adivinhando o número correto.

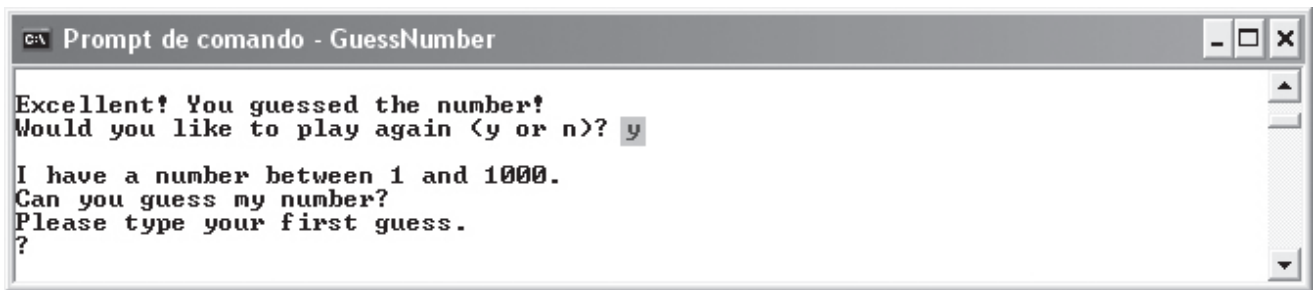


Figura 1.7 Jogando novamente.



Figura 1.8 Saindo do jogo.

Executando um aplicativo C++ utilizando GNU C++ com Linux

Para esse test-drive, assumimos que você sabe copiar os exemplos para seu diretório inicial. Consulte seu instrutor se tiver qualquer dúvida em relação a como copiar os arquivos para o sistema Linux. Além disso, para as figuras nesta seção, utilizamos uma formatação em negrito para indicar a entrada de usuário necessária a cada passo. O prompt no shell em nosso sistema utiliza o caractere til (~) para representar o diretório inicial e cada prompt termina com o caractere de cifrão (\$). O prompt irá variar entre sistemas Linux.

1. **Localizando o aplicativo concluído.** A partir de um shell do Linux, mude para o diretório do aplicativo `GuessNumber` completado (Figura 1.9) digitando

```
cd Examples/ch01/GuessNumber/GNU_Linux
```

depois pressione *Enter*. O comando `cd` é utilizado para alterar diretórios.

2. **Compilando o aplicativo `GuessNumber`.** Para executar um aplicativo no compilador GNU C++, ele deve ser primeiro compilado digitando

```
g++ GuessNumber.cpp -o GuessNumber
```

como na Figura 1.10. O comando anterior compila o aplicativo e produz um arquivo executável chamado `GuessNumber`.

3. **Executando o aplicativo `GuessNumber`.** Para executar o arquivo executável `GuessNumber`, digite `./GuessNumber` no próximo prompt, então pressione *Enter* (Figura 1.11).
4. **Inserindo sua primeira suposição.** O aplicativo exibe "Please type your first guess.", então exibe um ponto de interrogação (?) como um prompt na próxima linha (Figura 1.11). No prompt, insira **500** (Figura 1.12). [Nota: Esse é o mesmo aplicativo que modificamos e no qual fizemos o test-drive para Windows, mas as saídas poderiam variar, com base no compilador em uso.]

```
~$ cd examples/ch01/GuessNumber/GNU_Linux
~/examples/ch01/GuessNumber/GNU_Linux$
```

Figura 1.9 Alterando o diretório do aplicativo `GuessNumber` depois de efetuar login com sua conta Linux.

```
~/examples/ch01/GuessNumber/GNU_Linux$ g++ GuessNumber.cpp -o GuessNumber
~/examples/ch01/GuessNumber/GNU_Linux$
```

Figura 1.10 Compilando o aplicativo `GuessNumber` com o comando `g++`.

```
~/examples/ch01/GuessNumber/GNU_Linux$ ./GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
?
```

Figura 1.11 Executando o aplicativo `GuessNumber`.

```
~/examples/ch01/GuessNumber/GNU_Linux$ ./GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
?
```

Figura 1.12 Inserindo uma suposição inicial.

5. **Inserindo outra suposição.** O aplicativo exibe "Too high. Try again.", o que significa que o valor que você inseriu é maior que o número que o aplicativo escolheu como a suposição correta (Figura 1.12). No próximo prompt, insira **250** (Figura 1.13). Dessa vez o aplicativo exibe "Too low. Try again.", porque o valor que você inseriu é menor que a suposição correta.
6. **Inserindo suposições adicionais.** Continue o jogo (Figura 1.14) inserindo valores até você adivinhar o número correto. Quando adivinhar a resposta, o aplicativo exibirá "Excellent! You guessed the number!" (Figura 1.14).
7. **Jogando novamente ou fechando o aplicativo.** Depois de adivinhar o número correto, o aplicativo pergunta se você gostaria de ir para outro jogo. No prompt "Would you like to play again (y or n)?", inserir o caractere **y** faz com que o aplicativo escolha um novo número e exiba a mensagem "Please enter your first guess." seguida por um prompt de ponto de interrogação (Figura 1.15) para que você possa fazer sua primeira suposição no novo jogo. Inserir o caractere **n** fecha o aplicativo e o retorna para o diretório do aplicativo no shell (Figura 1.16). Toda vez que você executar esse aplicativo desde o início (isto é, *Passo 3*), ele escolherá os mesmos números para você adivinhar.

```
~/examples/ch01/GuessNumber/GNU_Linux$ ./GuessNumber
I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
? 250
Too low. Try again.
?
```

Figura 1.13 Inserindo uma segunda suposição e recebendo feedback.

```

Too low. Try again.
? 375
Too low. Try again.
? 437
Too high. Try again.
? 406
Too high. Try again.
? 391
Too high. Try again.
? 383
Too low. Try again.
? 387
Too high. Try again.
? 385
Too high. Try again.
? 384

Excellent! You guessed the number.
Would you like to play again (y or n)?

```

Figura 1.14 Inserindo suposições adicionais e adivinhando o número correto.

```

Excellent! You guessed the number.
Would you like to play again (y or n)? y

I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
?

```

Figura 1.15 Jogando novamente.

```

Excellent! You guessed the number.
Would you like to play again (y or n)? n

~/examples/ch01/GuessNumber/GNU_Linux$

```

Figura 1.16 Saindo do jogo.

1.17 Estudo de caso de engenharia de software: introdução à tecnologia de objetos e à UML (obrigatório)

Agora vamos começar nossa primeira introdução à orientação a objetos, uma maneira natural de pensar o mundo e escrever programas de computador. Os capítulos 1–7, 9 e 13 terminam com uma breve seção “Estudo de caso de engenharia de software”, em que apresentamos uma introdução passo a passo à orientação a objetos. Nosso objetivo aqui é ajudar você a desenvolver uma maneira orientada a objetos de pensar e apresentar-lhe a **Unified Modeling Language™ (UML™)** — uma linguagem gráfica que permite às pessoas que projetam sistemas de software orientados a objetos utilizar uma notação-padrão da indústria para representá-los.

Nesta seção obrigatória, introduzimos a terminologia e os conceitos básicos da orientação a objetos. As seções opcionais nos capítulos 2–7, 9 e 13 apresentam projeto e implementação orientados a objetos do software para um sistema de caixa automática (*automated teller machine* – ATM) simples. As seções “Estudo de caso de engenharia de software” no final dos capítulos 2–7

- analisam um típico documento de requisitos que descreve um sistema de software (o ATM) a ser construído;
- determinam os objetos necessários para implementar esse sistema;

- determinam os atributos que os objetos terão;
- determinam os comportamentos que esses objetos exibirão;
- especificam como os objetos interagem para atender os requisitos de sistema.

As seções “Estudo de caso de engenharia de software” no final dos capítulos 9 e 13 modificam e aprimoram o design apresentado nos capítulos 2–7. O Apêndice G contém uma implementação funcional e completa em C++ do sistema ATM orientado a objetos.

Embora nosso estudo de caso seja uma versão reduzida de um problema de nível industrial, abrangemos, apesar disso, muitas práticas comuns da indústria. Você experimentará uma introdução sólida ao projeto orientado a objetos com a UML. Além disso, aperfeiçoará suas habilidades de leitura de código fazendo um passeio pela completa implementação em C++ do ATM cuidadosamente escrita e bem-documentada.

Conceitos básicos da tecnologia de objeto

Iniciamos nossa introdução à orientação a objetos com uma terminologia-chave. Onde quer que você olhe no mundo real, você vê **objetos** — pessoas, animais, plantas, carros, aviões, edifícios, computadores e assim por diante. Os humanos pensam em termos de objetos. Telefones, casas, sinais de trânsito, fornos de microonda e sistemas de refrigeração à água são apenas mais alguns objetos que vemos ao nosso redor todos os dias.

Às vezes dividimos objetos em duas categorias: animados e inanimados. Os objetos animados são, em certo sentido, objetos ‘vivos’ — eles se movem e fazem coisas. Por outro lado, os objetos inanimados não se movem por conta própria. Objetos de ambos os tipos, porém, têm algumas coisas em comum. Todos eles têm **atributos** (por exemplo, tamanho, forma, cor e peso) e todos exibem **comportamentos** (por exemplo, uma bola rola, rebate, infla e murcha; o bebê chora, dorme, engatinha, anda e pisca; um carro acelera, freia e desvia; uma toalha absorve água). Estudaremos os tipos de atributos e comportamentos dos objetos de software.

Os humanos aprendem sobre os objetos existentes estudando seus atributos e observando seus comportamentos. Diferentes objetos podem ter atributos semelhantes e podem exibir comportamentos semelhantes. É possível fazer comparações, por exemplo, entre bebês e adultos, e entre humanos e chimpanzés.

O **projeto orientado a objetos** (*object-oriented design – OOD*) modela software em termos semelhantes àqueles que as pessoas utilizam para descrever objetos do mundo real. Ele tira proveito de relacionamentos de classe, em que os objetos de certa classe, como uma classe de veículos, têm as mesmas características — carros, caminhões e patins têm muito em comum. O OOD tira proveito dos relacionamentos de **herança**, em que novas classes de objetos são derivadas absorvendo-se características de classes existentes e adicionando-se características únicas dessas mesmas classes. Um objeto da classe ‘conversível’ certamente tem as características da classe mais geral ‘automóvel’, porém, mais especificamente, seu teto se abre e se fecha.

O projeto orientado a objetos fornece uma maneira natural e intuitiva de visualizar o processo de planejamento do software — a saber, modelar objetos por seus atributos, comportamentos e inter-relacionamentos, assim como descrevemos objetos do mundo real. A OOD também modela a comunicação entre objetos. Assim como as pessoas trocam mensagens entre si (por exemplo, um sargento manda um soldado permanecer em atenção), os objetos também se comunicam via mensagens. Um objeto conta bancária pode receber uma mensagem para reduzir seu saldo em certa quantia porque o cliente retirou essa quantia em dinheiro.

O OOD **encapsula** (isto é, empacota) atributos e **operações** (comportamentos) em objetos — os atributos e as operações de um objeto estão intimamente ligados. Os objetos têm a propriedade de **ocultamento de informações**. Isso significa que os objetos podem saber como se comunicar com outros por meio de **interfaces** bem-definidas, mas normalmente eles não têm permissão para saber como os outros objetos são implementados — os detalhes de implementação são ocultados dentro dos próprios objetos. Na verdade, podemos dirigir um carro, por exemplo, sem conhecer os detalhes de como motores, transmissões, freios e sistemas de escapamento funcionam internamente — contanto que saibamos utilizar o acelerador, o freio, a direção e assim por diante. O ocultamento de informações, como veremos, é crucial à boa engenharia de software.

Linguagens como C++ são **orientadas a objeto**. A programação nessa linguagem é chamada **programação orientada a objetos** (*object-oriented programming – OOP*) e permite aos programadores de computador implementar um projeto orientado a objetos como um sistema de software funcional. Linguagens como o C, por outro lado, são **procedurais**; portanto, a programação tende a ser **orientada para a ação**. No C, a unidade de programação é a **função**. No C++, a unidade de programação é a **classe** a partir da qual objetos são, por fim, **instanciados** (um termo da POO para ‘criados’). As classes C++ contêm funções que implementam operações e dados que implementam atributos.

Programadores de C concentram-se em escrever funções. Os programadores agrupam as ações que realizam alguma tarefa comum em funções e agrupam funções para formar programas. Os dados são certamente importantes em C, mas a visualização é que os dados existem principalmente em suporte das ações que as funções realizam. Os **verbos** em uma especificação de sistema ajudam o programador de C a determinar o conjunto de funções que trabalharão juntas para implementar o sistema.

Classes, membros de dados e funções-membro

Os programadores C++ concentram-se em criar seus próprios **tipos definidos pelo usuário** chamados **classes**. Cada classe contém dados, bem como o conjunto de funções que manipula esses dados e fornece serviços para **clientes** (isto é, outras classes ou funções que utilizam a classe). Os componentes de dados de uma classe são chamados de **membros de dados**. Por exemplo, uma classe conta bancária poderia incluir o número e o saldo de uma conta. Os componentes de função de uma classe são chamados de **funções-membro** (em geral, chamados de **métodos** em outras linguagens de programação orientada a objetos como o Java). Por exemplo, uma classe

conta bancária poderia incluir funções-membro para fazer um depósito (aumentando o saldo), fazer uma retirada (diminuindo o saldo) e consultar o saldo atual. O programador utiliza tipos predefinidos (e outros tipos definidos pelo usuário) como os ‘blocos de construção’ para construir novos tipos definidos pelo usuário (classes). Os **substantivos** em uma especificação de sistema ajudam o programador em C++ a determinar o conjunto de classes a partir da qual os objetos que são criados trabalham juntos para implementar o sistema.

As classes estão para os objetos como as plantas arquitetônicas estão para as casas — uma classe é um ‘plano’ para construir um objeto da classe. Assim como podemos construir muitas casas a partir de uma planta, podemos instanciar (criar) muitos objetos a partir de uma classe. Você não pode fazer refeições na cozinha de uma planta; isso só é possível em uma cozinha real. Você não pode dormir no quarto de uma planta arquitetônica; você só pode dormir no quarto de uma casa.

As classes podem ter relacionamentos com outras classes. Por exemplo, em um projeto orientado a objetos de um banco, a classe ‘caixa de banco’ precisa se relacionar com outras classes, como a classe ‘cliente’, a classe ‘gaveta de dinheiro’, a classe ‘cofre’ etc. Esses relacionamentos são chamados de **associações**.

Empacotar software como classes possibilita que os sistemas de software futuros **reutilizem** as classes. Grupos de classes relacionadas são freqüentemente empacotados como **componentes** reutilizáveis. Assim como corretores de imóveis costumam dizer que os três fatores mais importantes que afetam o preço dos imóveis são ‘localização, localização e localização’, as pessoas na comunidade de desenvolvimento de software costumam dizer que os três fatores mais importantes que afetam o futuro do desenvolvimento de software são ‘reutilização, reutilização e reutilização’.



Observação de engenharia de software 1.4

A reutilização de classes existentes ao construir novas classes e programas economiza tempo, dinheiro e esforço. A reutilização também ajuda os programadores a construir sistemas mais confiáveis e eficientes, porque classes e componentes existentes costumam passar por extensos testes, depurações e ajustes de desempenho.

De fato, com a tecnologia de objetos, você pode construir grande parte do novo software necessário combinando classes existentes, exatamente como fabricantes de automóveis combinam partes intercambiáveis. Cada nova classe que você cria terá o potencial de se tornar um valioso ativo de software que você e outros programadores podem reutilizar para acelerar e aprimorar a qualidade de seus esforços futuros no desenvolvimento de software.

Introdução à análise e projeto orientados a objetos (Object-oriented analysis and design – OOAD)

Logo você estará escrevendo programas em C++. Como criar o código para seus programas? Talvez, como muitos programadores iniciantes, você simplesmente ligará seu computador e começará a digitar. Essa abordagem pode funcionar para pequenos programas (como os apresentados nos primeiros capítulos do livro), mas e se você fosse contratado para criar um sistema de software para controlar milhares de caixas automáticos para um banco importante? Ou suponha que você vá trabalhar em uma equipe de 1.000 desenvolvedores de software para construir um sistema de última geração para o controle de tráfego aéreo dos Estados Unidos. Para projetos tão grandes e complexos, você não pode simplesmente sentar e começar a escrever programas.

Para criar as melhores soluções, você deve seguir um processo detalhado para **analisar** os **requisitos** do seu projeto (isto é, determinar *o que* o sistema deve fazer) e desenvolver um **design** que atenda esses requisitos (isto é, decidir *como* o sistema deve fazê-lo). Idealmente, você passaria por esse processo e revisaria cuidadosamente o design (ou teria seu design revisado por outros profissionais de software) antes de escrever qualquer código. Se esse processo envolve analisar e projetar o sistema de um ponto de vista orientado a objetos, ele é chamado de **processo de análise e projeto orientados a objetos (object-oriented analysis and design – OOAD)**. Programadores experientes sabem que análise e design podem poupar muitas horas ajudando a evitar uma abordagem de desenvolvimento de sistemas mal planejada que tem de ser abandonada no meio de sua implementação, possivelmente desperdiçando tempo, dinheiro e esforço consideráveis.

O OOAD é o termo genérico para o processo de análise de um problema e desenvolvimento de uma abordagem para resolvê-lo. Pequenos problemas como os discutidos nesses primeiros poucos capítulos não exigem um processo exaustivo de OOAD. Pode ser suficiente, antes de começarmos a escrever código C++, escrever um **pseudocódigo** — um modo informal baseado em texto de expressar a lógica do programa. Na realidade, o pseudocódigo não é uma linguagem de programação, mas pode ser utilizado como um esboço de orientação ao escrever o código. Introduzimos o pseudocódigo no Capítulo 4.

Uma vez que os problemas e os grupos de pessoas que os resolvem aumentam em tamanho, os métodos OOAD rapidamente tornam-se mais adequados do que o pseudocódigo. Idealmente, um grupo deve estabelecer um acordo comum sobre um processo rigorosamente definido para resolver seu problema e sobre uma maneira uniforme de comunicar os resultados desse processo para os outros. Embora existam muitos processos OOAD diferentes, uma única linguagem gráfica para comunicar os resultados de *qualquer* processo OOAD veio a ser amplamente utilizada. Essa linguagem, conhecida como Unified Modeling Language (UML), foi desenvolvida em meados da década de 1990 sob a direção inicial de três metodologistas de software: Grady Booch, James Rumbaugh e Ivar Jacobson.

História da UML

Na década de 1980, um número crescente de organizações começou a utilizar POO para construir seus aplicativos e desenvolveu-se a necessidade de um processo OOAD padrão. Muitos metodologistas — inclusive Booch, Rumbaugh e Jacobson — produziram e promoveram individualmente processos separados para satisfazer essa necessidade. Cada processo tinha sua própria notação ou ‘linguagem’ (na forma de diagramas gráficos), para transportar os resultados de análise e design.

Por volta do início de 1990, diferentes organizações e até divisões dentro de uma mesma organização estavam utilizando seus próprios processos e notações. Ao mesmo tempo, essas organizações também queriam utilizar ferramentas de software que suportassem seus processos particulares. Os fornecedores de software acharam difícil fornecer ferramentas para tantos processos. Claramente, uma notação-padrão e processos-padrão eram necessários.

Em 1994, James Rumbaugh associou-se a Grady Booch na Rational Software Corporation (agora uma divisão da IBM) e os dois começaram a trabalhar para unificar seus populares processos. Eles logo se associaram a Ivar Jacobson. Em 1996, o grupo liberou as primeiras versões da UML para a comunidade de engenharia de software e solicitaram feedback. Por volta da mesma época, uma organização conhecida como **Object Management Group™ (OMG™)** solicitou sugestões para uma linguagem de modelagem comum. O OMG (www.omg.org) é uma organização sem fins lucrativos que promove a padronização de tecnologias orientadas a objetos publicando diretrizes e especificações, como a UML. Várias corporações — entre elas HP, IBM, Microsoft, Oracle e Rational Software — já haviam reconhecido a necessidade de uma linguagem de modelagem comum. Em resposta à solicitação de propostas do OMG, essas empresas formaram o **UML Partners** — o consórcio que desenvolveu a UML versão 1.1 e a sugeriu para o OMG. O OMG aceitou a proposta e, em 1997, assumiu a responsabilidade pela manutenção e revisão constantes da UML. Em março de 2003, o OMG lançou a UML versão 1.5. A versão 2 da UML — que foi adotada e estava no processo de finalização quando escrevíamos esta publicação — marca a primeira revisão importante desde a versão 1.1 do padrão de 1997. Muitos livros, ferramentas de modelagem e especialistas da indústria já estão utilizando a UML versão 2, então apresentamos a terminologia e a notação da UML versão 2 por todo este livro.

O que é a UML?

A Unified Modeling Language é agora o esquema de representação gráfica mais amplamente utilizado para modelar sistemas orientados a objetos. Ela de fato unificou os vários esquemas de notação populares. Aqueles que projetam sistemas usam a linguagem (na forma de diagramas) para modelar seus sistemas, como fazemos por todo este livro.

Um recurso atraente da UML é sua flexibilidade. A UML é **extensível** (isto é, capaz de ser aprimorada com novos recursos) e é independente de qualquer processo OOAD particular. Os modeladores de UML são livres para utilizar vários processos ao projetar sistemas, mas agora todos os desenvolvedores podem expressar seus projetos com um conjunto-padrão de notações gráficas.

A UML é uma linguagem gráfica complexa rica em recursos. Em nossas seções “Estudo de caso de engenharia de software” sobre o desenvolvimento do software de um caixa automático (ATM), apresentamos um subconjunto conciso e simples desses recursos. Então utilizamos esse subconjunto para guiá-lo pela primeira experiência de design com a UML, destinada a programadores iniciantes em tecnologia orientada a objetos em um curso de programação do primeiro ou segundo semestre.

Esse estudo de caso foi cuidadosamente desenvolvido sob a orientação de revisores acadêmicos e profissionais. Esperamos que você se divirta trabalhando nele. Se tiver qualquer dúvida, entre em contato conosco em deitel@deitel.com. Responderemos prontamente.

Recursos da UML na Internet e Web

Para obter informações adicionais sobre a UML, consulte os seguintes sites Web. Para sites sobre UML adicionais, consulte os recursos na Internet e Web listados no fim da Seção 2.8.

www.uml.org

Esta página de recursos da UML do Object Management Group (OMG) fornece documentos de especificação da UML e outras tecnologias orientadas a objetos.

www.ibm.com/software/rational/uml

Esta é a página de recurso da UML da IBM Rational — a sucessora da Rational Software Corporation (a empresa que criou a UML).

Leituras recomendadas

Muitos livros sobre UML já foram publicados. Os seguintes livros recomendados fornecem informações sobre o projeto orientado a objetos com a UML.

- Arlow, J., and I. Neustadt. *UML and the unified process: practical object-oriented analysis and design*. Londres: Pearson Education Ltd., 2002.
- Fowler, M. *UML distilled, third edition: a brief guide to the standard object modeling language*. Boston: Addison-Wesley, 2004.
- Rumbaugh, J., I. Jacobson e G. Booch. *The unified modeling language user guide*. Reading, MA: Addison-Wesley, 1999.

Para livros adicionais sobre a UML, consulte a lista de leituras recomendadas no final da Seção 2.8 ou visite www.amazon.com ou www.bn.com. A IBM Rational, antes a Rational Software Corporation, também fornece uma lista de leituras recomendadas de livros sobre UML em www.ibm.com/software/rational/info/technical/books.jsp.

Seção 1.17 Exercícios de revisão

- 1.1** Liste três exemplos de objetos do mundo real que não mencionamos. Para cada objeto, liste vários atributos e comportamentos.
- 1.2** Pseudocódigo é _____.
 - a) outro termo para OOAD
 - b) uma linguagem de programação utilizada para exibir diagramas de UML

- c) uma maneira informal de expressar a lógica do programa
- d) um esquema de representação gráfica para modelar sistemas orientados a objetos

1.3 A UML é utilizada principalmente para _____.

- a) testar sistemas orientados a objetos
- b) projetar sistemas orientados a objetos
- c) implementar sistemas orientados a objetos
- d) a e b são alternativas corretas

Respostas aos exercícios de revisão da Seção 1.17

1.1 [Nota: As respostas podem variar.] a) Os atributos de uma televisão incluem o tamanho da tela, o número de cores que ela pode exibir, seu canal atual e volume atual. Uma televisão liga e desliga, muda de canais, exibe vídeo e reproduz sons. b) Os atributos de uma cafeteira incluem o volume máximo de água que ela pode conter, o tempo necessário para fazer um bule de café e a temperatura da chapa sob o bule. Uma cafeteira liga e desliga, faz e esquentar café. c) Os atributos de uma tartaruga incluem a idade, tamanho do casco e peso. Uma tartaruga caminha, protege-se dentro de seu casco, sai de seu casco e alimenta-se de vegetais.

1.2 c.

1.3 b.

1.18 Síntese

Este capítulo introduziu conceitos básicos de hardware e software e explorou o papel do C++ no desenvolvimento de aplicativos cliente/servidor distribuídos. Você estudou a história da Internet e da World Wide Web. Discutimos os diferentes tipos de linguagens de programação, sua história e as linguagens de programação que são mais amplamente utilizadas. Discutimos também a C++ Standard Library, que contém classes e funções reutilizáveis que ajudam os programadores em C++ a criar programas C++ portáteis.

Apresentamos conceitos básicos da tecnologia de objeto, incluindo classes, objetos, atributos, comportamentos, encapsulamento e herança. Você também aprendeu a história e o propósito da UML — a linguagem gráfica padrão da indústria para modelar sistemas de software.

Você aprendeu os passos típicos para criar e executar um aplicativo C++. Por fim, você fez o ‘test-drive’ de um aplicativo C++ de exemplo semelhante aos tipos de aplicativo que você aprenderá a programar neste livro.

No próximo capítulo, você criará seus primeiros aplicativos C++. Você verá vários exemplos que demonstram como exibir mensagens de programas na tela e obter informações do usuário no teclado para processamento. Analisamos e explicamos cada exemplo para ajudá-lo a facilitar a introdução na programação C++.

1.19 Recursos na Web

Esta seção fornece muitos recursos da Web que serão úteis para você à medida que aprende a linguagem C++. Os sites incluem recursos C++, ferramentas de desenvolvimento C++ para alunos e profissionais e alguns links para jogos divertidos construídos com C++. Esta seção também lista seus próprios sites Web onde você pode localizar downloads e recursos associados com este livro. Você encontrará recursos na Web adicionais no Apêndice I.

Sites Web da Deitel & Associates

www.deitel.com/books/cppHTP5/index.html

O site do C++ *How to Program, Fifth Edition* da Deitel & Associates. Aqui o leitor encontrará links para os exemplos do livro (também incluídos no CD que acompanha o livro) e outros recursos, como nossos guias *DIVE INTO™* gratuitos que o ajudam a começar a aprender os vários ambientes de desenvolvimento integrado (*integrated development environments – IDEs*) C++.

www.deitel.com

Consulte o site da Deitel & Associates para obter atualizações, correções e recursos adicionais de todas as publicações da Deitel.

www.deitel.com/newsletter/subscribe.html

Visite esse site para assinar o boletim de correio eletrônico *Deitel Buzz Online* a fim de seguir o programa de publicações da Deitel & Associates.

www.prenhall.com/deitel

O site da Prentice Hall para as publicações da Deitel. Aqui você encontrará informações detalhadas sobre nossos produtos, capítulos de exemplo e *Companion Web Sites* que contém recursos específicos do livro e do capítulo para alunos e instrutores.

Compiladores e ferramentas de desenvolvimento

www.thefreecountry.com/developercity/ccompilers.shtml

Esse site lista compiladores C e C++ gratuitos para uma variedade de sistemas operacionais.

msdn.microsoft.com/visualc

O site *Microsoft Visual C++* fornece informações de produto, resumos, materiais suplementares e informações de compra do compilador Visual C++.

www.borland.com/bcppbuilder

Esse é um link para o *C++ Builder da Borland*. Uma versão de linha de comando gratuita está disponível para download.

www.compilers.net

O *Compilers.net* foi projetado para ajudar usuários a localizar compiladores.

developer.intel.com/software/products/compilers/cwin/index.htm

Um download de avaliação do *compilador Intel C++* está disponível nesse site.

www.kai.com/C_plus_plus

Esse site oferece uma versão de avaliação de 30 dias do *compilador C++ da Kai*.

www.symbian.com/developer/development/cppdev.html

O Symbian fornece um C++ Developer's Pack e links para vários recursos, incluindo código e ferramentas de desenvolvimento para programadores em C++ que implementam aplicativos móveis para o sistema operacional Symbian, que é popular em dispositivos como telefones celulares.

Recursos

www.hal9k.com/cug

O *site C/C++ Users Group (CUG)* contém recursos, periódicos, shareware e freeware sobre C++.

www.devx.com

O *DevX* é um recurso abrangente para programadores que fornece as últimas notícias, ferramentas e técnicas para várias linguagens de programação. O *C++ Zone* oferece dicas, fóruns de discussão, ajuda técnica e newsletters on-line.

www.acm.org/crossroads/xrds3-2/ovp32.html

O *site Association for Computing Machinery (ACM)* oferece uma abrangente listagem de recursos C++, incluindo textos recomendados, periódicos e revistas, padrões publicados, newsletters, FAQs e newsgroups.

www.accu.informika.ru/resources/public/terse/cpp.htm

O *site Association of C & C++ Users (ACCU)* contém links para tutoriais de C++, artigos, informações de desenvolvedor, discussões e sinopses de livros.

www.cuj.com

O *C/C++ User's Journal* é uma revista on-line que contém artigos, tutoriais e downloads. O site apresenta notícias sobre C++, fóruns e links para informações sobre ferramentas de desenvolvimento.

www.research.att.com/~bs/homepage.html

Esse é o site de Bjarne Stroustrup, projetista da linguagem de programação C++. Esse site fornece uma lista de recursos do C++, FAQs e outras informações C++ úteis.

Jogos

www.codearchive.com/list.php?go=0708

Esse site tem vários jogos em C++ disponíveis para download.

www.mathtools.net/C_C_/Games/

Esse site inclui links para numerosos jogos construídos com C++. O código-fonte para a maioria dos jogos está disponível para download.

www.gametutorials.com/Tutorials/GT/GT_Pg1.htm

Esse site tem tutoriais sobre programação de jogo em C++. Cada tutorial inclui uma descrição do jogo e uma lista dos métodos e funções utilizadas no tutorial.

www.forum.nokia.com/main/0,6566,050_20,00.html

Visite esse site da Nokia para aprender a utilizar C++ para programar jogos de alguns dispositivos sem fio da Nokia.

Resumo

- Os vários dispositivos que abrangem um sistema de computador são referidos como hardware.
- Os programas que executam em um computador são referidos como software.
- Um computador é capaz de realizar computações e tomar decisões lógicas a velocidades de milhões e (até bilhões) de vezes mais rápido que o homem.
- Os computadores processam dados sob o controle de conjuntos de instruções chamados programas de computador, que orientam o computador por meio de conjuntos ordenados de ações especificadas por programadores de computador.
- A unidade de entrada é a seção 'receptora' do computador. Ela obtém informações de dispositivos de entrada e as coloca à disposição das outras unidades para processamento.

- A unidade de saída é a seção de ‘envio’ do computador. Ela recebe informações (dados e programas de computador) de vários dispositivos de entrada e coloca essas informações à disposição de outras unidades de modo que as informações possam ser processadas.
- A unidade de memória é a seção de armazenamento de relativamente baixa capacidade e acesso rápido do computador. Ela retém informações que foram inseridas pela unidade de entrada, disponibilizando-as imediatamente para processamento quando necessário, e retém informações que já foram processadas até que essas informações possam ser colocadas em dispositivos de saída pela unidade de saída.
- A unidade lógica e aritmética (ALU) é a seção de ‘fabricação’ do computador. É responsável por realizar cálculos e tomar decisões.
- A unidade central de processamento (CPU) é a seção ‘administrativa’ do computador. Ela coordena e supervisiona a operação das outras seções.
- A unidade de armazenamento secundário é a seção de armazenamento de alta capacidade e longo prazo do computador. Normalmente, os programas ou dados que não estão sendo utilizados por outras unidades são colocados em dispositivos de armazenamento secundários (por exemplo, discos) até que sejam necessários novamente, possivelmente horas, dias, meses ou mesmo anos mais tarde.
- Os sistemas operacionais foram desenvolvidos para ajudar a tornar o uso de computadores mais conveniente.
- A multiprogramação envolve o compartilhamento dos recursos de um computador entre os trabalhos que disputam sua atenção, de modo que esses pareçam executar simultaneamente.
- Com a computação distribuída, a computação de uma organização é distribuída em redes para os sites em que o trabalho da organização é realizado.
- Qualquer computador pode entender diretamente apenas sua própria linguagem de máquina, que geralmente consiste em strings de números instruindo os computadores a realizar suas operações mais elementares.
- Abreviações em inglês formam a base das linguagens assembly. Os programas tradutores chamados assemblers convertem programas de linguagem assembly em linguagem de máquina.
- Compiladores traduzem programas de linguagem de alto nível em programas de linguagem de máquina. Linguagens de alto nível (como C++) contêm palavras inglesas e notações matemáticas convencionais.
- Os programas interpretadores executam diretamente os programas de linguagem de alto nível, eliminando a necessidade de compilá-los em linguagem de máquina.
- O C++ evoluiu a partir do C, que evoluiu de duas linguagens de programação, BCPL e B.
- O C++ é uma extensão do C desenvolvida por Bjarne Stroustrup no início da década de 1980 na Bell Laboratories. O C++ aprimora a linguagem C e fornece capacidades para programação orientada a objetos.
- Objetos são componentes de software reutilizáveis que modelam itens do mundo real. Utilizar uma abordagem de projeto e implementação modulares e orientadas a objetos pode tornar grupos de desenvolvimento de software mais produtivos do que com as técnicas de programação anteriores.
- Os programas C++ consistem em partes chamadas classes e funções. Você pode programar cada parte de que possa precisar para formar um programa C++. Mas a maioria dos programadores em C++ tira proveito das ricas coleções de classes e funções existentes na C++ Standard Library.
- O Java é utilizado para criar conteúdo dinâmico e interativo para páginas Web, desenvolver aplicativos corporativos, aprimorar funcionalidade de servidor Web, fornecer aplicativos para dispositivos de consumo popular e muito mais.
- A linguagem FORTRAN (FORMula TRANslator) foi desenvolvida pela IBM Corporation em meados da década de 1950 para aplicativos científicos e de engenharia que requerem complexos cálculos matemáticos.
- A linguagem COBOL (COMmon Business Oriented Language) foi desenvolvida no final da década de 1950 por um grupo de fabricantes de computador e usuários de computadores do governo e da indústria. O COBOL é utilizado principalmente para aplicativos comerciais que exigem manipulação de dados precisa e eficiente.
- A linguagem Ada foi desenvolvida sob o patrocínio do Departamento da Defesa dos Estados Unidos durante a década de 1970 e início da de 1980. A Ada fornece multitarefa, que permite aos programadores especificar que muitas atividades devem ocorrer paralelamente.
- A linguagem BASIC (Beginner’s All-Purpose Symbolic Instruction Code) de programação foi desenvolvida em meados de 1960 no Dartmouth College como uma linguagem para escrever programas simples. O principal propósito da BASIC foi familiarizar os iniciantes com as técnicas de programação.
- A linguagem Visual Basic da Microsoft foi introduzida no início da década de 1990 para simplificar o processo de desenvolvimento de aplicativos Microsoft Windows.
- A Microsoft tem uma estratégia de escopo corporativo para integrar a Internet e a Web em aplicativos de computador. Essa estratégia é implementada na plataforma .NET da Microsoft.
- As três principais linguagens de programação da plataforma .NET são Visual Basic .NET (baseada na linguagem BASIC original), Visual C++ .NET (baseada em C++) e C# (uma nova linguagem baseada em C++ e Java que foi desenvolvida especificamente para a plataforma .NET).
- Os desenvolvedores em .NET podem escrever componentes de software em sua linguagem preferida e, assim, formar aplicativos combinando esses componentes com componentes escritos em qualquer linguagem .NET.

- Em geral, os sistemas C++ consistem em três partes: um ambiente de desenvolvimento de programa, a linguagem e a C++ Standard Library.
- Os programas C++, em geral, passam por seis fases: edição, pré-processamento, compilação, linkagem, carregamento e execução.
- Nomes de arquivo de código-fonte C++ costumam terminar com as extensões `.cpp`, `.cxx`, `.cc` ou `.C`.
- Um programa pré-processador executa automaticamente antes de a fase de conversão do compilador iniciar. O pré-processador C++ obedece a comandos chamados diretivas de pré-processador, que indicam que certas manipulações devem ser realizadas no programa antes da compilação.
- Em geral, o código-objeto produzido pelo compilador C++ contém ‘lacunas’ por causa das referências a funções e dados definidos em outra parte. Um linker vincula o código-objeto com o código das funções ausentes a fim de produzir uma imagem executável (sem partes ausentes).
- O carregador pega a imagem executável a partir de disco e a transfere para a memória para execução.
- A maioria dos programas em C++ realiza entrada e/ou saída de dados. Os dados são freqüentemente inseridos a partir de `cin` (o fluxo de entrada padrão) que, em geral, é o teclado, mas `cin` pode ser redirecionado a partir de outro dispositivo. A saída dos dados é, em geral, enviada para `cout` (o fluxo de saída padrão), que, normalmente, é a tela do computador, mas `cout` pode ser redirecionado para outro dispositivo. O fluxo `cerr` é utilizado para exibir mensagens de erro.
- A Unified Modeling Language (UML) é uma linguagem gráfica que permite que as pessoas construam sistemas para representar seus projetos orientados a objetos em uma notação comum.
- O projeto orientado a objetos (*object-oriented design* – OOD) modela componentes de software em termos de objetos do mundo real. Ela tira proveito dos relacionamentos de classe, em que os objetos de certa classe têm as mesmas características. Ele também tira proveito de relacionamentos de herança, em que as novas classes de objetos criadas são derivadas absorvendo-se características de classes existentes e adicionando-se características únicas dessas mesmas classes. O OOD encapsula dados (atributos) e funções (comportamento) em objetos — os dados e funções de um objeto são intimamente conectados.
- Os objetos têm a propriedade de ocultamento de informações — normalmente os objetos não têm permissão de saber como outros objetos são implementados.
- A programação orientada a objetos (*object-oriented programming* – OOP) permite aos programadores implementar projetos orientados a objetos como sistemas funcionais.
- Os programadores em C++ criam seus próprios tipos definidos pelo usuário chamados classes. Toda classe contém dados (conhecidos como membros de dados) e o conjunto de funções (conhecido como funções-membro) que manipula esses dados e fornece serviços para clientes.
- As classes podem ter relacionamentos com outras classes. Esses relacionamentos são chamados associações.
- Empacotar software como classes possibilita que os sistemas de software futuros reutilizem as classes. Grupos de classes relacionadas são freqüentemente empacotados como componentes reutilizáveis.
- Uma instância de uma classe é chamada objeto.
- Com a tecnologia de objeto, os programadores podem construir grande parte do software que será necessária combinando partes intercambiáveis e padronizadas chamadas classes.
- O processo de analisar e projetar um sistema a partir de um ponto de vista orientado a objetos é chamado análise e projeto orientados a objetos (*object-oriented analysis and design* – OOAD).

Terminologia

ação	C padrão ANSI/ISSO	computação cliente/servidor
.NET Framework Class Library da Microsoft	C#	computação distribuída
abordagem de código ativo	C++	computação pessoal
Ada	C++ padrão ANSI/ISSO	computador
American National Standards Institute (ANSI)	C++ Standard Library	conteúdo dinâmico
análise	carregador	dados
análise e design de sistemas estruturados	classe	decisão
análise e projeto orientados a objetos (object-oriented analysis and design – OOAD)	cliente	dependente de máquina
assembler	COBOL (COmmon Business Oriented Language)	depurar
associação	código-fonte	design
atributo de um objeto	código-objeto	diretivas de pré-processador
BASIC (Beginner’s All-Purpose Symbolic Instruction Code)	compartilhamento de tempo	dispositivo de entrada
Booch, Grady	compilador	dispositivo de saída
C	componente	documento de requisitos
	comportamento de um objeto	editor
		encapsular

entrada/saída (E/S)	linker	projeto orientado a objetos (object-oriented design – OOD)
erros de tempo de execução	membro de dados	pseudocódigo
estação de trabalho	memória	Rational Software Corporation
extensível	memória básica	redes locais (local area networks – LANs)
fase de carga	memória principal	reutilização de software
fase de compilação	método	Rumbaugh, James
fase de edição	MFC (Microsoft Foundation Classes)	servidor de arquivos
fase de execução	multiprocessador	sistema operacional
fase de link	multiprogramação	software
fase de pré-processamento	multitarefa	supercomputador
FORTRAN (FORMula TRANslator)	multithreading	tarefa
função	Object Management Group (OMG)	throughput
função-membro	objeto	tipo definido pelo usuário
hardware	ocultamento de informações	tradução
herança	operação	unidade aritmética e lógica (ALU)
imagem executável	plataforma	unidade central de processamento (central processing unit – CPU)
independente de máquina	plataforma .NET	unidade de armazenamento secundária
instanciar	plataforma de hardware	unidade de entrada
interface	portável	unidade de memória
International Organization for Standardization (ISO)	processamento em lote	unidade de saída
Internet	programa de computador	unidade lógica
interpretador	programa tradutor	Unified Modeling Language (UML)
Jacobson, Ivar	programação estruturada	Visual Basic .NET
Java	programação orientada a objetos (object-oriented programming – OOP)	Visual C++ .NET
linguagem assembly	programação procedural	World Wide Web
linguagem de alto nível	programador de computador	
linguagem de máquina		

Exercícios de revisão

1.1 Preencha as lacunas em cada uma das seguintes sentenças:

- A empresa que popularizou a computação pessoal foi _____.
- O computador que tornou a computação pessoal legítima nos negócios e na indústria foi _____.
- Os computadores processam dados sob o controle de conjuntos de instruções chamados _____ de computador.
- As seis principais unidades lógicas do computador são _____, _____, _____, _____, _____ e _____.
- As três classes de linguagens discutidas no capítulo são _____, _____ e _____.
- Os programas que traduzem programas de linguagem de alto nível em linguagem de máquina são chamados _____.
- C é amplamente conhecido como a linguagem de desenvolvimento do sistema operacional _____.
- A linguagem _____ foi desenvolvida por Wirth para ensinar programação estruturada.
- O Departamento da Defesa dos Estados Unidos desenvolveu a linguagem Ada com uma capacidade chamada _____, que permite aos programadores especificar que muitas atividades podem ocorrer em paralelo.

1.2 Preencha as lacunas em cada uma das seguintes frases sobre o ambiente C++.

- Os programas C++ normalmente são digitados em um computador utilizando um programa _____.
- Em um sistema C++, um programa _____ executa antes de a fase de conversão do compilador iniciar.
- O programa _____ combina a saída do compilador com várias funções de biblioteca para produzir uma imagem executável.
- O programa _____ transfere a imagem executável de um programa C++ do disco para a memória.

1.3 Preencha as lacunas de cada uma das sentenças a seguir (com base na Seção 1.17):

- Os objetos têm a propriedade de _____ — embora os objetos possam saber comunicar-se entre si por meio de interfaces bem definidas, normalmente não têm permissão de saber como outros objetos são implementados.
- Os programadores em C++ se concentram na criação de _____, que contêm membros de dados e as funções-membro que manipulam esses membros de dados e fornecem serviços para clientes.
- As classes podem ter relacionamentos com outras classes. Esses relacionamentos são chamados _____.
- O processo de analisar e projetar um sistema de um ponto de vista orientado a objetos é chamado _____.

- e) O OOD também tira proveito de relacionamentos de _____, em que novas classes de objetos são derivadas absorvendo-se características de classes existentes e, em seguida, adicionando-se características únicas dessas mesmas classes.
- f) _____ é uma linguagem gráfica que permite que as pessoas que projetam sistemas de software utilizem uma notação-padrão da indústria para representá-las.
- g) O tamanho, forma, cor e peso de um objeto são considerados _____ do objeto.

Respostas dos exercícios de revisão

- 1.1** a) Apple. b) IBM Personal Computer. c) programas. d) unidade de entrada, unidade de saída, unidade de memória, unidade de aritmética e lógica, unidade central de processamento, unidade de armazenamento secundária. e) linguagens de máquina, linguagens assembly, linguagens de alto nível. f) compiladores. g) UNIX h) Pascal. i) multitarefa.
- 1.2** a) editor. b) pré-processador. c) linker. d) carregador.
- 1.3** a) ocultamento de informações. b) classes. c) associações. d) análise e projeto orientados a objetos (*object-oriented analysis and design* – OOAD). e) herança. f) Unified Modeling Language (UML). g) atributos.

Exercícios

- 1.4** Categorize cada um dos itens seguintes como hardware ou software:
 - a) CPU
 - b) Compilador C++
 - c) ALU
 - d) Pré-processador C++
 - e) unidade de entrada
 - f) um programa de editor
- 1.5** Por que você poderia querer escrever um programa em uma linguagem independente de máquina em vez de uma linguagem dependente de máquina? Por que uma linguagem dependente de máquina talvez fosse mais apropriada para escrever certos tipos de programas?
- 1.6** Preencha as lacunas em cada uma das seguintes afirmações:
 - a) Qual unidade lógica do computador recebe informações de fora do computador para utilização pelo computador? _____.
 - b) O processo de instrução do computador para resolver problemas específicos é chamado _____.
 - c) Que tipo de linguagem de computador utiliza abreviações semelhantes ao inglês para instruções de linguagem de máquina? _____.
 - d) Qual unidade lógica do computador envia informações que já foram processadas pelo computador para vários dispositivos de modo que as informações possam ser utilizadas fora do computador? _____.
 - e) Qual unidade lógica do computador retém informações? _____.
 - f) Qual unidade lógica do computador realiza cálculos? _____.
 - g) Qual unidade lógica do computador toma decisões lógicas? _____.
 - h) O nível de linguagem de computador mais conveniente para o programador escrever programas rápida e facilmente é _____.
 - i) A única linguagem que um computador entende diretamente é chamada _____ desse computador.
 - j) Qual unidade lógica do computador coordena as atividades de todas as outras unidades lógicas? _____.
- 1.7** Por que hoje se concentra tanta atenção na programação orientada a objetos em geral e no C++ em particular?
- 1.8** Por que você talvez prefira experimentar um erro fatal em vez de um erro não fatal? Por que você poderia preferir que seu programa sofresse um erro fatal em vez de um erro não fatal?
- 1.9** Dê uma resposta breve a cada uma das seguintes perguntas:
 - a) Por que esse texto discute a programação estruturada além da programação orientada a objetos?
 - b) Quais são os passos típicos (mencionados no texto) de um processo de projeto orientado a objetos?
 - c) Que tipos de mensagens as pessoas enviam entre si?
 - d) Os objetos enviam entre si mensagens por meio de interfaces bem-definidas. Que interfaces um rádio de carro (objeto) apresenta para seu usuário (um objeto pessoa)?
- 1.10** Você provavelmente está usando no seu pulso um dos tipos de objetos mais comuns do mundo — um relógio. Discuta como cada um dos seguintes termos e conceitos se aplica à noção de um relógio: objeto, atributos, comportamentos, classe, herança (considere, por exemplo, um despertador), abstração, modelagem, mensagens, encapsulamento, interface, ocultamento de informações, membros de dados e funções-membro.



*Que há num nome simples?
que chamamos rosa,
que com outro nome
seria tão perfumado.*
William Shakespeare

*Quando preciso tomar uma
decisão, sempre pergunto: "O
que seria mais divertido?"*
Peggy Walker

*"Tome mais chá", a Lebre de
Março disse para Alice, muito
sinceramente. "Eu ainda não
bebi nada", Alice respondeu
em um tom ofendido: "então
não posso tomar mais." "Você
quis dizer que não pode tomar
menos", disse Leirão: "é muito
mais fácil tomar mais do que
não tomar nada."*
Lewis Carroll

*Pensamentos elevados devem
ter uma linguagem elevada.*
Aristóфанes

Introdução à programação em C++

OBJETIVOS

Neste capítulo, você aprenderá:

- A escrever programas de computador simples em C++.
- A escrever instruções de entrada e saída simples.
- A utilizar tipos fundamentais.
- Os conceitos básicos de memória de computador.
- A utilizar operadores aritméticos.
- A precedência dos operadores aritméticos.
- A escrever instruções de tomada de decisão simples.

- 2.1 Introdução
- 2.2 Primeiro programa C++: imprimindo uma linha de texto
- 2.3 Modificando nosso primeiro programa C++
- 2.4 Outro programa C++: adicionando inteiros
- 2.5 Conceitos de memória
- 2.6 Aritmética
- 2.7 Tomada de decisão: operadores de igualdade e operadores relacionais
- 2.8 Estudo de caso de engenharia de software: examinando o documento de requisitos de ATM (opcional)
- 2.9 Síntese

Resumo | Terminologia | Exercícios de revisão | Respostas dos exercícios de revisão | Exercícios

2.1 Introdução

Introduzimos agora a programação C++, que facilita uma abordagem disciplinada ao design de programa. A maioria dos programas C++ que você estudará neste livro processa informações e exibe os resultados. Neste capítulo, apresentamos cinco exemplos que demonstram como seus programas exibem mensagens e obtêm informações do usuário para processamento. Os três primeiros exemplos simplesmente exibem mensagens na tela. O quarto é um programa que obtém dois números de um usuário, calcula sua soma e exibe o resultado. A discussão que o acompanha mostra como realizar vários cálculos aritméticos e salvar seus resultados para uso posterior. O quinto exemplo demonstra os fundamentos de tomada de decisão mostrando como comparar dois números e, então, exibir mensagens com base nos resultados de comparação. Analisamos cada programa uma linha por vez para ajudar a facilitar a introdução à programação C++. Para ajudar a aplicar as habilidades aprendidas aqui, fornecemos vários problemas de programação nos exercícios do capítulo.

2.2 Primeiro programa C++: imprimindo uma linha de texto

O C++ utiliza notações que podem parecer estranhas aos não-programadores. Agora consideramos um programa simples que imprime uma linha de texto (Figura 2.1). Esse programa ilustra vários recursos importantes da linguagem C++. Consideramos cada linha detalhadamente.

As linhas 1 e 2

```
// Figura 2.1: fig02_01.cpp
// Programa de impressão de texto.
```

começam com `//`, indicando que o restante de cada linha é um **comentário**. Os programadores inserem comentários para documentar programas e também para ajudar as pessoas a ler e a entender programas. Os comentários não fazem com que o computador realize qualquer ação quando o programa está em execução — eles são ignorados pelo compilador C++ e não fazem com que qualquer código-objeto de linguagem de máquina seja gerado. O comentário Programa de impressão de texto descreve o propósito do programa. Um comentário que inicia com `//` é chamado **comentário de uma única linha** porque termina no final da linha atual. [Nota: Os programadores em C++ também podem utilizar o estilo C em que um comentário — possivelmente contendo muitas linhas — inicia com o par de caracteres `/*` e termina com `*/`.]

```
1 // Figura 2.1: fig02_01.cpp
2 // Programa de impressão de texto.
3 #include <iostream> // permite que o programa gere saída de dados na tela
4
5 // a função main inicia a execução do programa
6 int main()
7 {
8     std::cout << "Welcome to C++!\n"; // exibe a mensagem
9
10    return 0; // indica que o programa terminou com sucesso
11
12 } // fim da função main
```

Welcome to C++!

Figura 2.1 Programa de impressão de texto.



Boa prática de programação 2.1

Cada programa deve iniciar com um comentário que descreve o propósito do programa, autor, data e hora. (Não mostramos o autor, data e hora nos programas deste livro porque essas informações seriam redundantes.)

A linha 3

```
#include <iostream> // permite que o programa gere saída de dados na tela
```

é uma **diretiva de pré-processador**, que é uma mensagem para o pré-processador C++ (introduzido na Seção 1.14). As linhas que iniciam com # são processadas pelo pré-processador antes de o programa ser compilado. Essa linha instrui o pré-processador a incluir no programa o conteúdo do **arquivo de cabeçalho de fluxo de entrada/saída <iostream>**. Esse arquivo deve ser incluído em qualquer programa que realize a saída de dados na tela ou a entrada de dados a partir do teclado utilizando entrada/saída de fluxo no estilo C++. O programa na Figura 2.1 gera saída de dados na tela, como veremos em breve. Discutimos os arquivos de cabeçalho em mais detalhes no Capítulo 6 e explicamos o conteúdo de `iostream` no Capítulo 15.



Erro comum de programação 2.1

Esquecer de incluir o arquivo de cabeçalho <iostream> em um programa que realiza entrada de dados a partir do teclado ou saída de dados na tela faz com que o compilador emita uma mensagem de erro, porque o compilador não pode reconhecer referências aos componentes de fluxo (por exemplo, `cout`).

A linha 4 é simplesmente uma linha em branco. Os programadores utilizam linhas em branco, caracteres de espaço em branco e de tabulação (isto é, ‘tabs’) para facilitar a leitura de programas. Juntos, esses caracteres são conhecidos como **espaços em branco**. Os caracteres de espaço em branco são normalmente ignorados pelo compilador. Neste e nos vários capítulos seguintes, discutimos as convenções de utilização de caracteres de espaço em branco para aprimorar a legibilidade do programa.



Boa prática de programação 2.2

Utilize linhas em branco e caracteres de espaço em branco para aprimorar a legibilidade do programa.

A linha 5

```
// a função main inicia a execução do programa
```

é outro comentário de uma única linha que indica que a execução de programa inicia na próxima linha.

A linha 6

```
int main()
```

é uma parte de cada programa C++. Os parênteses depois de `main` indicam que `main` é um bloco de construção de programa chamado **função**. Os programas C++, em geral, consistem em uma ou mais funções e classes (como você aprenderá no Capítulo 3). Todo programa deve ter exatamente uma função `main`. A Figura 2.1 contém somente uma função. Os programas C++ começam executando na função `main`, mesmo se `main` não for a primeira função no programa. A palavra-chave `int` à esquerda de `main` indica que `main` ‘retorna’ um valor do tipo inteiro (número inteiro). Uma **palavra-chave** é uma palavra em código que é reservada pelo C++ para uma utilização específica. A lista completa de palavras-chave C++ pode ser encontrada na Figura 4.3. Explicaremos o que significa uma função ‘retornar um valor’ quando demonstrarmos como criar suas próprias funções na Seção 3.5 e quando estudarmos as funções em maior profundidade no Capítulo 6. Por enquanto, simplesmente inclua a palavra-chave `int` à esquerda de `main` em cada um de seus programas.

A **chave esquerda**, `,` (linha 7) deve começar o **corpo** de cada função. Uma **chave direita** correspondente, `,` (linha 12) deve terminar o corpo de cada função. A linha 8

```
std::cout << "Welcome to C++!\n"; // exibe a mensagem
```

instrui o computador a **realizar uma ação** — a saber, imprimir a **string** de caracteres contida entre as aspas duplas. Uma string às vezes é chamada de **string de caractere**, **mensagem** ou **string literal**. Nós nos referimos a caracteres entre aspas duplas genericamente como **strings**. Os caracteres de espaço em branco em strings não são ignorados pelo compilador.

A linha 8 inteira, incluindo `std::cout`, o **operador <<**, a string `"Welcome to C++!\n"` e o **ponto-e-vírgula** `( ; )`, é chamada de **instrução**. Toda instrução em C++ deve terminar com um ponto-e-vírgula (também conhecido como **terminador de instrução**). As diretivas de pré-processador (como, `#include`) não terminam com um ponto-e-vírgula. A saída e a entrada em C++ são realizadas com **fluxos** de caracteres. Portanto, quando a instrução precedente é executada, ela envia o fluxo de caracteres `Welcome to C++!\n` para o **objeto de fluxo de saída padrão** — `std::cout` — que, normalmente, é ‘conectado’ à tela. Discutimos os muitos recursos `std::cout` detalhadamente no Capítulo 15, “Entrada/saída de fluxo”.

Note que colocamos `std::` antes de `cout`. Isso é necessário quando utilizamos nomes trazidos no programa pela diretiva de pré-processador `#include <iostream>`. A notação `std::cout` especifica que estamos utilizando um nome, nesse caso, `cout`, que pertence ao ‘namespace’ `std`. Os nomes `cin` (o fluxo de entrada padrão) e `cerr` (o fluxo de erro padrão) — introduzidos no Capítulo 1 — também pertencem ao namespace `std`. Os namespaces são um recurso avançado do C++ que discutimos em profundidade no Capítulo 24, “Outros tópicos”. Por enquanto, você deve simplesmente lembrar de incluir `std::` antes de cada menção de `cout`, `cin` e `cerr` em um programa. Isso pode ser incômodo — na Figura 2.13, introduzimos a declaração `using`, que permitirá omitirmos `std::` antes de cada uso de um nome no namespace `std`.

O operador `<<` é referido como o **operador de inserção de fluxo**. Quando esse programa executa, o valor à direita do operador, o **operando** direito, é inserido no fluxo de saída. Note que o operador aponta na direção de onde entram os dados. Em geral, os caracteres do operando direito são impressos exatamente como aparecem entre as aspas duplas. Note, porém, que os caracteres `\n` não são impressos na tela. A barra invertida (`\`) é chamada **caractere de escape**. Ela indica que um caractere ‘especial’ deve ser enviado para a saída. Quando uma barra invertida é encontrada em uma string de caracteres, o próximo caractere é combinado com a barra invertida para formar uma **seqüência de escape**. A seqüência de escape `\n` significa **nova linha**. Isso faz com que o **cursor** (isto é, o indicador da posição atual na tela) se mova para o começo da próxima linha na tela. Algumas outras seqüências de escape comuns são listadas na Figura 2.2.



Erro comum de programação 2.2

*Omitir o ponto-e-vírgula no fim de uma instrução C++ é um erro de sintaxe. (Novamente, as diretivas de pré-processador não terminam em um ponto-e-vírgula.) A **sintaxe** de uma linguagem de programação especifica as regras da criação de um programa adequado nessa linguagem. Um **erro de sintaxe** ocorre quando o compilador encontra o código que viola as regras da linguagem do C++ (isto é, sua sintaxe). O compilador normalmente emite uma mensagem de erro para ajudar o programador a localizar e corrigir o código incorreto. Erros de sintaxe também são chamados **erros de compilador**, **erros em tempo de compilação** ou **erros de compilação**, porque o compilador os detecta durante a fase de compilação. Não será possível executar seu programa até você corrigir todos os erros de sintaxe nele. Como você verá, alguns erros de compilação não são erros de sintaxe.*

A linha 10

```
return 0; // indica que o programa terminou com sucesso
```

é uma das várias maneiras que utilizaremos para **sair de uma função**. Quando a instrução `return` é utilizada no fim de `main`, como mostrado aqui, o valor 0 indica que o programa terminou com sucesso. No Capítulo 6, discutimos as funções em detalhes, e as razões de incluir essa instrução se tornarão claras. Por enquanto, simplesmente inclua essa instrução em cada programa; caso contrário, o compilador pode produzir um aviso em alguns sistemas. A chave direita, `}`, (linha 12) indica o fim da função `main`.



Boa prática de programação 2.3

Muitos programadores tornam o último caractere impresso por uma função um caractere de nova linha (`\n`). Isso assegura que a função deixará o cursor de tela posicionado no começo de uma nova linha. Convenções dessa natureza encorajam a reusabilidade de software — um objetivo-chave no desenvolvimento de software.



Boa prática de programação 2.4

Recue o corpo inteiro de cada função um nível dentro das chaves que delimitam o corpo da função. Isso faz a estrutura funcional de um programa destacar-se e ajuda a tornar o programa mais fácil de ler.



Boa prática de programação 2.5

Defina uma convenção para o tamanho de recuo preferido, então aplique-a uniformemente. A tecla Tab pode ser utilizada para criar recuos, mas as paradas de tabulação podem variar. Recomendamos utilizar paradas de tabulação de 1/4 de polegada (0,63 cm) ou (preferivelmente) três espaços para formar um nível de recuo.

Seqüência de escape	Descrição
<code>\n</code>	Nova linha. Posiciona o cursor de tela para o início da próxima linha.
<code>\t</code>	Tabulação horizontal. Move o cursor de tela para a próxima parada de tabulação.
<code>\r</code>	Retorno de carro. Posiciona o cursor da tela no início da linha atual; não avança para a próxima linha.
<code>\a</code>	Alerta. Aciona o aviso sonoro do sistema.
<code>\\</code>	Barras invertidas. Utilizadas para imprimir um caractere de barra invertida.
<code>\'</code>	Aspas simples. Utilizadas para imprimir um único caractere de aspas simples.
<code>\"</code>	Aspas duplas. Utilizadas para imprimir um caractere de aspas duplas.

Figura 2.2 Seqüências de escape.

2.3 Modificando nosso primeiro programa C++

Esta seção continua nossa introdução à programação C++ com dois exemplos, mostrando como modificar o programa na Figura 2.1 para imprimir texto em uma linha utilizando múltiplas instruções e imprimir texto em várias linhas utilizando uma única instrução.

Imprimindo uma única linha de texto com múltiplas instruções

Wellcome to C++! pode ser impresso de várias maneiras. Por exemplo, a Figura 2.3 realiza a inserção de fluxo em múltiplas instruções (linhas 8–9), mas ainda produz a mesma saída que o programa da Figura 2.1. [Nota: Deste ponto em diante, utilizamos um fundo cinza na tabela de código para destacar os recursos-chave que cada programa introduz.] Cada inserção de fluxo retoma a impressão a partir de onde a anterior parou. A primeira inserção de fluxo (linha 8) imprime a palavra Wellcome seguida por um espaço; e a segunda inserção de fluxo (linha 9) começa a imprimir a mesma linha logo depois do espaço. Em geral, o C++ permite ao programador expressar instruções de várias maneiras.

Imprimindo múltiplas linhas de texto com uma única instrução

Uma única instrução pode imprimir múltiplas linhas utilizando caracteres de nova linha, como na linha 8 da Figura 2.4. Toda vez que a sequência de escape \n (nova linha) for encontrada no fluxo de saída, o cursor de tela é posicionado no começo da linha seguinte. Para obter uma linha em branco em sua saída, coloque dois caracteres de nova linha um após o outro, como na linha 8.

```

1 // Figura 2.3: fig02_03.cpp
2 // Imprimindo uma linha de texto com múltiplas instruções.
3 #include <iostream> // permite que o programa gere saída de dados na tela
4
5 // a função main inicia a execução do programa
6 int main()
7 {
8     std::cout << "Wellcome ";
9     std::cout << "to C++!\n";
10
11     return 0; // indica que o programa terminou com sucesso
12
13 } // fim da função main

```

Wellcome to C++!

Figura 2.3 Imprimindo uma linha de texto com múltiplas instruções.

```

1 // Figura 2.4: fig02_04.cpp
2 // Imprimindo múltiplas linhas de texto com uma única instrução.
3 #include <iostream> // permite que o programa gere saída de dados na tela
4
5 // a função main inicia a execução do programa
6 int main()
7 {
8     std::cout << "Wellcome\n\nto\n\nC++!\n";
9
10     return 0; // indica que o programa terminou com sucesso
11
12 } // fim da função main

```

Wellcome
to

C++!

Figura 2.4 Imprimindo múltiplas linhas de texto com uma única instrução.

2.4 Outro programa C++: adicionando inteiros

Nosso próximo programa utiliza o objeto de fluxo de entrada `std::cin` e o **operador de extração de fluxo**, `>>`, para obter dois inteiros digitados por um usuário no teclado, calcula a soma desses valores e gera a saída do resultado utilizando `std::cout`. A Figura 2.5 mostra o programa e os exemplos de entradas e saídas. Observe que destacamos a entrada do usuário em negrito.

Os comentários nas linhas 1 e 2

```
// Figura 2.5: fig02_05.cpp
// Programa de adição que exibe a soma de dois números.
```

declaram o nome do arquivo e o propósito do programa. A diretiva de pré-processador C++

```
#include <iostream> // permite ao programa realizar entrada e saída
```

na linha 3 inclui o conteúdo do arquivo de cabeçalho `iostream` no programa.

O programa inicia a execução com função `main` (linha 6). A chave esquerda (linha 7) marca o começo de corpo do `main` e a chave direita correspondente (linha 25) marca o fim de `main`.

As linhas 9–11

```
int number1; // primeiro inteiro a adicionar
int number2; // segundo inteiro a adicionar
int sum; // soma de number1 e number2
```

são **declarações**. Os identificadores `number1`, `number2` e `sum` são os nomes de **variáveis**. Uma variável é uma posição na memória do computador onde um valor pode ser armazenado para utilização por um programa. Essas declarações especificam que as variáveis `number1`, `number2` e `sum` são dados de tipo `int`, o que significa que essas variáveis armazenarão valores **inteiros**, isto é, números inteiros como 7, -11, 0 e 31.914. Todas as variáveis devem ser declaradas com um nome e um tipo de dados antes que possam ser utilizadas em um programa. Diversas variáveis do mesmo tipo podem ser declaradas em uma declaração ou em múltiplas declarações. Poderíamos ter declarado todas as três variáveis em uma declaração assim:

```
int number1, number2, sum;
```

```
1 // Figura 2.5: fig02_05.cpp
2 // Programa de adição que exibe a soma de dois números.
3 #include <iostream> // permite ao programa realizar entrada e saída
4
5 // a função main inicia a execução do programa
6 int main()
7 {
8     // declarações de variável
9     int number1; // primeiro inteiro a adicionar
10    int number2; // segundo inteiro a adicionar
11    int sum; // soma de number1 e number2
12
13    std::cout << "Enter first integer: "; // solicita dados ao usuário
14    std::cin >> number1; // lê primeiro inteiro inserido pelo usuário em number1
15
16    std::cout << "Enter second integer: "; // solicita dados ao usuário
17    std::cin >> number2; // lê segundo inteiro inserido pelo usuário em number2
18
19    sum = number1 + number2; // adiciona os números; armazena o resultado em sum
20
21    std::cout << "Sum is " << sum << std::endl; // exibe sum; termina a linha
22
23    return 0; // indica que o programa terminou com sucesso
24
25 }
```

```
Enter first integer: 45
Enter second integer: 72
Sum is 117
```

Figura 2.5 O programa de adição que exibe a soma de dois inteiros inseridos a partir do teclado.

Isso torna o programa menos legível e impede que forneçamos comentários que descrevem o propósito de cada variável. Se mais de um nome é declarado em uma declaração (como mostrado aqui), os nomes são separados por vírgulas (,). Isso é referido como uma **lista separada por vírgulas**.



Boa prática de programação 2.6

Coloque um espaço depois de cada vírgula (,) para tornar os programas mais legíveis.



Boa prática de programação 2.7

Alguns programadores preferem declarar cada variável em uma linha separada. Esse formato permite inserção fácil de um comentário descritivo ao lado de cada declaração.

Em breve, discutiremos o tipo de dados `double` para especificar números reais e o tipo de dados `char` para especificar dados de caractere. Os números reais são números com pontos de fração decimal, como 3,4, 0,0 e -11,19. Uma variável `char` pode armazenar apenas uma única letra minúscula, uma única maiúscula, um único dígito ou um único caractere especial (por exemplo, \$ ou *). Tipos como `int`, `double` e `char` são freqüentemente chamados **tipos fundamentais**, **tipos primitivos** ou **tipos predefinidos**. Os nomes dos tipos fundamentais são palavras-chave e, portanto, devem aparecer em letras minúsculas. O Apêndice C contém a lista completa de tipos fundamentais.

Um nome variável (como `number1`) é qualquer **identificador** válido que não seja uma palavra-chave. Um identificador é uma série de caracteres consistindo em letras, dígitos e sublinhados (_) que não iniciam com um dígito. O C++ faz **distinção de letras maiúsculas e minúsculas** — letras minúsculas e maiúsculas são diferentes, então `a1` e `A1` são identificadores diferentes.



Dica de portabilidade 2.1

O C++ permite identificadores de qualquer comprimento, mas sua implementação do C++ pode impor algumas restrições sobre o comprimento de identificadores. Utilize identificadores de 31 caracteres ou menos para assegurar portabilidade.



Boa prática de programação 2.8

*Escolher identificadores significativos ajuda a fazer um programa **autodocumentado** — uma pessoa pode entender o programa simplesmente lendo-o em vez de ter de referir-se a manuais ou comentários.*



Boa prática de programação 2.9

Evite usar abreviações em identificadores. Isso aumenta a legibilidade do programa.



Boa prática de programação 2.10

Evite identificadores que iniciem com sublinhados e sublinhados duplos, porque os compiladores C++ podem utilizar nomes assim para seus próprios propósitos internamente. Isso impedirá que os nomes que você escolhe sejam confundidos com nomes escolhidos por compiladores.



Dica de prevenção de erro 2.1

Linguagens como C++ são 'alvos inconstantes'. À medida que elas evoluem, mais palavras-chave poderiam ser adicionadas à linguagem. Evite utilizar palavras 'sobrecarregadas de significado' como 'object' para identificadores. Ainda que 'object' não seja atualmente uma palavra-chave em C++, ela poderia vir a tornar-se uma; portanto, a compilação futura com novos compiladores poderia quebrar o código existente.

As declarações de variáveis podem ser colocadas quase em qualquer lugar em um programa, mas devem aparecer antes de suas variáveis correspondentes serem utilizadas no programa. Por exemplo, no programa da Figura 2.5, a declaração na linha 9

```
int number1; // primeiro inteiro a adicionar
```

poderia ter sido colocada imediatamente antes da linha 14

```
std::cin >> number1; // lê primeiro inteiro inserido pelo usuário em number1
```

a declaração na linha 10

```
int number2; // segundo inteiro a adicionar
```

poderia ter sido colocada imediatamente antes da linha 17

```
std::cin >> number2; // lê segundo inteiro inserido pelo usuário em number2
```

e a declaração na linha 11

```
int sum; // soma de number1 e number2
```

poderia ter sido colocada imediatamente antes da linha 19

```
sum = number1 + number2; // adiciona os números; armazena o resultado em sum
```



Boa prática de programação 2.11

Sempre coloque uma linha em branco entre uma declaração e instruções executáveis adjacentes. Isso faz com que as declarações se destaquem e contribui para a clareza do programa.



Boa prática de programação 2.12

Se preferir colocar declarações no começo de uma função, separe-as das instruções executáveis nessa função com uma linha em branco para destacar onde as declarações terminam e as instruções executáveis iniciam.

A linha 13

```
std::cout << "Enter first integer: "; // solicita dados ao usuário
```

imprime a string `Enter first integer:` (também conhecida como um literal string ou um **literal**) na tela. Essa mensagem é chamada **prompt** porque direciona o usuário para uma ação específica. Gostamos de pronunciar a instrução precedente como ‘`std::cout` *obtem* a string de caractere “Enter first integer: ”’. A linha 14

```
std::cin >> number1; // lê primeiro inteiro inserido pelo usuário em number1
```

utiliza o **objeto fluxo de entrada cin** (do namespace `std`) e o **operador de extração de fluxo**, `>>`, para obter um valor do teclado. Utilizar o operador de extração de fluxo com `std::cin` aceita a entrada de caractere a partir do fluxo de entrada padrão, que geralmente é o teclado. Gostamos de pronunciar a instrução anterior como ‘`std::cin` *fornece* um valor ao `number1`’ ou simplesmente ‘`std::cin` *fornece* `number1`’.



Dica de prevenção de erro 2.2

Os programas devem validar a correção de todos os valores de entrada para impedir que informações errôneas afetem os cálculos de um programa.

Quando o computador executa a instrução anterior, ele espera o usuário inserir um valor para a variável `number1`. O usuário responde digitando um inteiro (como caracteres), e, então, pressionando a tecla *Enter* (às vezes chamada de tecla *Return*) a fim de enviar os caracteres para o computador. O computador converte a representação de caractere do número em um inteiro e atribui (copia) esse número (ou **valor**) para a variável `number1`. Qualquer referência subsequente a `number1` nesse programa utilizará esse mesmo valor.

Os objetos de fluxo `std::cout` e `std::cin` facilitam a interação entre o usuário e o computador. Como essa interação assemelha-se a um diálogo, ela é frequentemente chamada de **computação conversacional** ou **computação interativa**.

A linha 16

```
std::cout << "Enter second integer: "; // solicita dados ao usuário
```

imprime `Enter second integer:` na tela, pedindo para o usuário executar uma ação. A linha 17

```
std::cin >> number2; // lê segundo inteiro inserido pelo usuário em number2
```

obtem um valor para a variável `number2` a partir do usuário.

A instrução de atribuição na linha 19

```
sum = number1 + number2; // adiciona os números; armazena o resultado em sum
```

calcula a soma das variáveis `number1` e `number2` e atribui o resultado à variável `sum` utilizando o **operador de atribuição** `=`. A instrução é lida como ‘`sum` *obtem* o valor de `number1 + number2`’. A maioria dos cálculos é realizada em instruções de atribuição. O operador `=` e o operador `+` são chamados de **operadores binários** porque cada um tem dois operandos. No caso do operador `+`, os dois operandos são `number1` e `number2`. No caso do operador `=` anterior, os dois operandos são `sum` e o valor da expressão `number1 + number2`.



Boa prática de programação 2.13

Coloque espaços de ambos os lados de um operador binário. Isso destaca o operador e torna o programa mais legível.

A linha 21

```
std::cout << "Sum is " << sum << std::endl; // exhibe sum; termina a linha
```

exibe a string de caractere `Sum is` seguida pelo valor numérico da variável `sum` seguida por `std::endl` — chamada **manipulador de fluxo**. O nome `endl` é uma abreviação de ‘end line’ e pertence ao namespace `std`. O manipulador de fluxo `std::endl` gera saída de um caractere de nova linha e, depois, ‘esvazia o buffer de saída’. Isso simplesmente significa que, em alguns sistemas em que as

saídas acumulam na máquina até haver conteúdo suficiente para ‘valer a pena’ a exibição na tela, `std::endl` força que todas as saídas acumuladas sejam exibidas nesse momento. Isso pode ser importante quando as saídas exigem que o usuário execute uma ação, como inserir dados.

Observe que a instrução precedente gera saída de múltiplos valores de diferentes tipos. O operador de inserção de fluxo ‘sabe’ gerar saída de cada tipo de dados. Utilizar múltiplos operadores de inserção de fluxo (`<<`) em uma única instrução é referido como **operações de inserção de fluxo de concatenação**, **encadeamento** ou **em cascata**. É desnecessário ter múltiplas instruções para gerar saída de múltiplas partes dos dados.

Os cálculos também podem ser feitos em instruções de saída. Poderíamos ter combinado as instruções nas linhas 19 e 21 na instrução

```
std::cout << "Sum is " << number1 + number2 << std::endl;
```

eliminando, assim, a necessidade da variável `sum`.

Um recurso poderoso do C++ é que os usuários podem criar seus próprios tipos de dados chamados classes (introduzimos essa capacidade no Capítulo 3 e a exploramos profundamente nos capítulos 9 e 10). Os usuários então podem ‘ensinar’ o C++ a realizar entrada e saída de valores desses novos tipos de dados utilizando os operadores `>>` e `<<` (isso é chamado **sobrecarga de operadores** — um tópico que exploramos no Capítulo 11).

2.5 Conceitos de memória

Os nomes de variável como `number1`, `number2` e `sum` correspondem na realidade às **posições** na memória do computador. Cada variável tem um nome, um tipo, um tamanho e um valor.

No programa de adição da Figura 2.5, quando a instrução

```
std::cin >> number1; // lê primeiro inteiro inserido pelo usuário em number1
```

na linha 14 é executada, os caracteres digitados pelo usuário são convertidos em um inteiro que é colocado em uma posição da memória à qual o nome `number1` foi atribuído pelo compilador C++. Suponha que o usuário insira o número 45 como o valor para `number1`. O computador colocará 45 na posição `number1`, como mostrado na Figura 2.6.

Sempre que um valor é colocado em uma posição da memória, o valor sobrescreve o valor anterior nessa posição; portanto, diz-se que é **destrutivo** colocar um novo valor em uma posição da memória.

Retornando ao nosso programa de adição, quando a instrução

```
std::cin >> number2; // lê segundo inteiro inserido pelo usuário em number2
```

na linha 17 é executada, suponha que o usuário insira o valor 72. Esse valor é colocado na posição `number2` e a memória aparece como mostrado na Figura 2.7. Observe que essas posições não são necessariamente adjacentes na memória.

Uma vez que o programa obteve os valores para `number1` e `number2`, ele soma esses valores e coloca a soma na variável `sum`. A instrução

```
sum = number1 + number2; // adiciona os números; armazena o resultado em sum
```

que realiza a adição também substitui qualquer valor que foi armazenado em `sum`. Isso ocorre quando a soma calculada de `number1` e `number2` é colocada na posição `sum` (sem considerar qual valor já pode estar em `sum`; esse valor é perdido). Depois de `sum` ser calculada, a memória aparece como na Figura 2.8. Observe que os valores de `number1` e `number2` aparecem exatamente como eram antes de ser utilizados no cálculo de `sum`. Esses valores foram utilizados, mas não destruídos, enquanto o computador realizou o cálculo. Portanto, quando um valor é lido de uma posição da memória, o processo é do tipo **não destrutivo**.



Figura 2.6 Posição da memória mostrando o nome e o valor da variável `number1`.

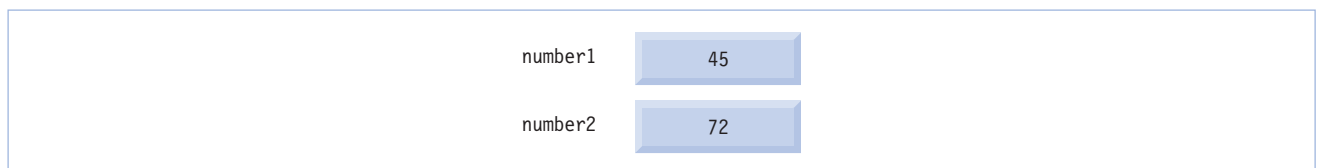


Figura 2.7 Posições da memória depois de armazenar os valores para `number1` e `number2`.

number1	45
number2	72
sum	117

Figura 2.8 Posições da memória depois de calcular e armazenar a sum de number1 e number2.

2.6 Aritmética

A maioria dos programas realiza cálculos aritméticos. A Figura 2.9 resume os **operadores aritméticos** do C++. Observe o uso de vários símbolos especiais não utilizados em álgebra. O **asterisco** (*) indica multiplicação; e o **sinal de porcentagem** (%) é o operador **módulo** que será discutido em breve. Os operadores aritméticos na Figura 2.9 são todos operadores binários, isto é, operadores que aceitam dois operandos. Por exemplo, a expressão `number1 + number2` contém o operador binário `+` e os dois operandos `number1` e `number2`.

A **divisão de inteiros** (isto é, aquela em que tanto o numerador como o denominador são inteiros) produz um quociente do tipo inteiro; por exemplo, a expressão `7 / 4` é avaliada como 1, e a expressão `17 / 5`, como 3. Observe que qualquer parte fracionária na divisão de inteiro é descartada (isto é, **truncada**) — nenhum arredondamento ocorre.

O C++ fornece o **operador módulo**, `%`, que fornece o resto da divisão de inteiros. O operador módulo pode ser utilizado somente com operandos inteiros. A expressão `x % y` produz o restante depois que `x` é dividido por `y`. Portanto, `7 % 4` produz 3 e `17 % 5` produz 2. Nos capítulos posteriores, discutimos muitas aplicações interessantes do operador módulo, tais como determinar se um número é um múltiplo de outro (um caso especial disso é determinar se um número é ímpar ou par).



Erro comum de programação 2.3

Tentar utilizar o operador módulo (%) com operandos não inteiros é um erro de compilação.

Expressões aritméticas em linha reta

As expressões aritméticas em C++ devem ser inseridas no computador em **linha reta**. Portanto, as expressões como ‘a dividido por b’ devem ser escritas como `a / b` de modo que todas as constantes, variáveis e operadores apareçam em uma sequência direta. A notação algébrica

$$\frac{a}{b}$$

em geral não é aceitável para compiladores, embora existam alguns pacotes de software de uso especial que suportem notação mais natural para expressões matemáticas complexas.

Parênteses para agrupar subexpressões

Os parênteses são utilizados em expressões C++ da mesma maneira que em expressões algébricas. Por exemplo, para multiplicar a vezes a quantidade `b + c`, escrevemos `a * ( b + c )`.

Regras de precedência de operadores

O C++ aplica os operadores em expressões aritméticas em uma sequência precisa determinada pelas seguintes **regras de precedência de operador**, que em geral são as mesmas que aquelas seguidas em álgebra:

Operação C++	Operador aritmético C++	Expressão algébrica	Expressão C++
Adição	<code>+</code>	$f + 7$	<code>f + 7</code>
Subtração	<code>-</code>	$p - c$	<code>p - c</code>
Multiplicação	<code>*</code>	bm ou $b \cdot m$	<code>b * m</code>
Divisão	<code>/</code>	x / y ou $\frac{x}{y}$ ou $x \div y$	<code>x / y</code>
Módulo	<code>%</code>	$r \bmod s$	<code>r % s</code>

Figura 2.9 Operadores aritméticos.

1. Os operadores em expressões contidas dentro de pares de parênteses são avaliados primeiro. Portanto, os *parênteses podem ser utilizados para forçar a ordem de avaliação a ocorrer em qualquer sequência desejada pelo programador*. Dizemos que os parênteses estão no nível ‘mais alto de precedência’. Em casos de **parênteses aninhados**, ou **embutidos**, como

$$((a + b) + c)$$

os operadores no par mais interno de parênteses são aplicados primeiro.

2. Operações de multiplicação, divisão e módulo são aplicadas em seguida. Se uma expressão contém várias operações de multiplicação, divisão e módulo, os operadores são aplicados da esquerda para a direita. Diz-se que a multiplicação, a divisão e o módulo estão no mesmo nível de precedência.
3. Operações de adição e de subtração são aplicadas por último. Se uma expressão contém várias operações de adição e subtração, os operadores são aplicados da esquerda para a direita. Adição e subtração também têm o mesmo nível de precedência.

O conjunto de regras de precedência de operadores define a ordem em que o C++ aplica operadores. Quando dizemos que certos operadores são aplicados da esquerda para a direita, estamos nos referindo à **associatividade** dos operadores. Por exemplo, na expressão

$$a + b + c$$

os operadores de adição (+) associam-se da esquerda para a direita, então $a + b$ é calculado primeiro, depois c é adicionado a essa soma para determinar o valor da expressão inteira. Veremos que alguns operadores se associam da direita para a esquerda. A Figura 2.10 resume essas regras de precedência de operador. Essa tabela será expandida à medida que os operadores adicionais do C++ forem introduzidos. Uma tabela completa de precedência está incluída no Apêndice A.

Exemplo de expressões algébricas e expressões em C++

Agora considere várias expressões à luz das regras de precedência de operadores. Cada exemplo lista uma expressão algébrica e sua expressão equivalente em C++. O seguinte é um exemplo de uma média aritmética de cinco termos:

Álgebra: $m = \frac{a + b + c + d + e}{5}$

C++: $m = (a + b + c + d + e) / 5;$

Os parênteses são exigidos porque a divisão tem precedência mais alta que a adição. A quantidade inteira $(a + b + c + d + e)$ deve ser dividida por 5. Se os parênteses são omitidos erroneamente, obtemos $a + b + c + d + e / 5$, que é incorretamente avaliado como

$$a + b + c + d + \frac{e}{5}$$

O seguinte é um exemplo da equação de uma linha reta:

Álgebra: $y = mx + b$

C++: $y = m * x + b;$

Nenhum parêntese é requerido. A multiplicação é aplicada primeiro porque a multiplicação tem uma precedência mais alta que a adição.

O exemplo a seguir contém operações de módulo (%), multiplicação, divisão, adição, subtração e atribuição:

Álgebra: $z = pr \% q + w / x - y$

C++: $z = p * r \% q + w / x - y;$



Operador(es)	Operação(ções)	Ordem de avaliação (precedência)
()	Parênteses	Avaliados primeiro. Se os parênteses estão aninhados, a expressão no par mais interno é avaliada primeiro. Se há vários pares de parênteses ‘no mesmo nível’ (isto é, não aninhados), eles são avaliados da esquerda para a direita.
*	Multiplicação	Avaliado em segundo lugar. Se houver vários, eles são avaliados da esquerda para a direita.
/	Divisão	
%	Módulo	
+	Adição	Avaliado por último. Se houver vários, eles são avaliados da esquerda para a direita.
-	Subtração	

Figura 2.10 Precedência de operadores aritméticos.

Os números dentro de círculos sob a instrução indicam a ordem em que o C++ aplica os operadores. A multiplicação, o módulo e a divisão são avaliados primeiro na ordem da esquerda para a direita (isto é, eles se associam da esquerda para a direita) porque eles têm precedência mais alta que a adição e a subtração. A adição e a subtração são aplicadas a seguir. Elas também são aplicadas da esquerda para a direita. Então, o operador de atribuição é aplicado.

Avaliação de um polinômio de segundo grau

Para desenvolver um melhor entendimento das regras de precedência de operadores, considere a avaliação de um polinômio de segundo grau ($y = ax^2 + bx + c$):

$y = a * x * x + b * x + c;$

6 1 2 4 3 5

Os números dentro de círculos sob a instrução indicam a ordem em que o Java aplica os operadores. Não há nenhum operador aritmético para exponenciação em C++, então representamos x^2 como $x * x$. Em breve discutiremos a função de biblioteca-padrão `pow` ('potência') que realiza exponenciação. Por causa de algumas questões sutis relacionadas com os tipos de dados exigidos por `pow`, adiamos uma explicação detalhada de `pow` até o Capítulo 6.



Erro comum de programação 2.4

Algumas linguagens de programação utilizam os operadores `**` ou `^` para representar a exponenciação. O C++ não suporta esses operadores de exponenciação; utilizá-los para exponenciação resulta em erros.

Suponha que as variáveis a , b , c e x no polinômio precedente de segundo grau sejam inicializadas da seguinte maneira: $a = 2$, $b = 3$, $c = 7$ e $x = 5$. A Figura 2.11 ilustra a ordem em que os operadores são aplicados.

Como em álgebra, é aceitável colocar parênteses desnecessários em uma expressão para tornar a expressão mais clara. Estes são chamados **parênteses redundantes**. Por exemplo, a instrução de atribuição precedente poderia estar entre parênteses como a seguir:

$y = (a * x * x) + (b * x) + c;$



Boa prática de programação 2.14

Utilizar parênteses redundantes em expressões aritméticas complexas pode tornar as expressões mais claras.

2.7 Tomada de decisão: operadores de igualdade e operadores relacionais

Esta seção introduz uma versão simples da **instrução `if`** do C++ que permite que um programa tome uma decisão com base na verdade ou falsidade de alguma **condição**. Se a condição for satisfeita, isto é, se a condição for verdadeira, a instrução no corpo da instrução `if`

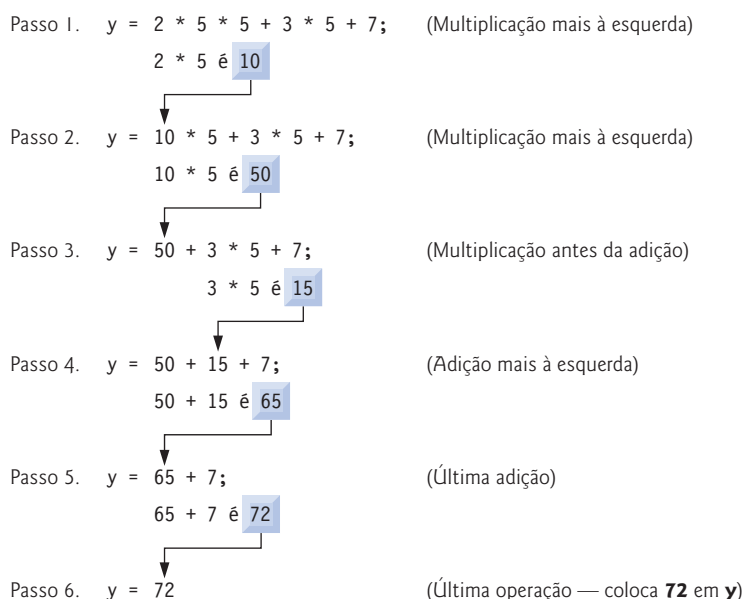


Figura 2.11 Ordem em que um polinômio de segundo grau é avaliado.

será executada. Se a condição não for satisfeita, isto é, se a condição for falsa, a instrução no corpo não será executada. Veremos um exemplo brevemente.

As condições em instruções `if` podem ser formadas utilizando **operadores de igualdade** e **operadores relacionais** resumidos na Figura 2.12. Todos os operadores relacionais têm o mesmo nível de precedência e se associam da esquerda para a direita. Ambos os operadores de igualdade têm o mesmo nível de precedência, que é mais baixo que aquele dos operadores relacionais, e eles se associam da esquerda para a direita.



Erro comum de programação 2.5

Ocorrerá um erro de sintaxe se algum dos operadores `==`, `!=`, `>=` e `<=` aparecer com espaços entre seu par de símbolos.



Erro comum de programação 2.6

Inverter a ordem do par de símbolos em algum dos operadores `!=`, `>=` e `<=` (escrevendo-os como `!=`, `=>` e `=<`, respectivamente) é, em geral, um erro de sintaxe. Em alguns casos, escrever `!=` como `=!` não será um erro de sintaxe, mas, quase certamente, um **erro de lógica** que tem um efeito em tempo de execução. Você entenderá a razão disso quando aprender sobre operadores lógicos no Capítulo 5. Um **erro de lógica fatal** faz com que um programa falhe e termine prematuramente. Um **erro de lógica não fatal** permite a um programa continuar executando, mas geralmente produz resultados incorretos.



Erro comum de programação 2.7

Confundir o operador de igualdade `==` com o operador de atribuição `=` resulta em erros de lógica. O operador de igualdade deve ser lido como ‘é igual a’, e o operador de atribuição deve ser lido como ‘obtem’ ou ‘obtem o valor de’ ou ‘recebe o valor de’. Algumas pessoas preferem ler o operador de igualdade como ‘duplo igual’. Como discutimos na Seção 5.9, confundir esses operadores pode não causar necessariamente um erro de sintaxe fácil de reconhecer, mas pode causar erros de lógica extremamente sutis.

O seguinte exemplo utiliza seis instruções `if` para comparar dois números inseridos pelo usuário. Se a condição em qualquer dessas instruções `if` for satisfeita, a instrução de saída associada com essa instrução `if` é executada. A Figura 2.13 mostra o programa e os diálogos de entrada/saída de três execuções de exemplo.

As linhas 6–8

```
using std::cout; // o programa utiliza cout
using std::cin;  // o programa utiliza cin
using std::endl; // o programa utiliza endl
```

são **declarações using** que eliminam a necessidade de repetir o prefixo `std::` como repetimos em programas anteriores. Uma vez que inserimos essas declarações `using`, podemos escrever `cout` em vez de `std::cout`, `cin` em vez de `std::cin` e `endl` em vez de `std::endl`, respectivamente, no restante do programa. [Nota: Deste ponto em diante no livro, cada exemplo contém uma ou mais declarações `using`.]

Operador algébrico de igualdade ou relacional padrão	Operador de igualdade ou relacional em C++	Exemplo de condição em C++	Significado da condição em C++
<i>Operadores relacionais</i>			
<code>></code>	<code>></code>	<code>x > y</code>	<code>x</code> é maior que <code>y</code>
<code><</code>	<code><</code>	<code>x < y</code>	<code>x</code> é menor que <code>y</code>
<code>≥</code>	<code>>=</code>	<code>x >= y</code>	<code>x</code> é maior que ou igual a <code>y</code>
<code>≤</code>	<code><=</code>	<code>x <= y</code>	<code>x</code> é menor que ou igual a <code>y</code>
<i>Operadores de igualdade</i>			
<code>=</code>	<code>==</code>	<code>x == y</code>	<code>x</code> é igual a <code>y</code>
<code>≠</code>	<code>!=</code>	<code>x != y</code>	<code>x</code> não é igual a <code>y</code>

Figura 2.12 Operadores de igualdade e operadores relacionais.

```

1 // Figura 2.13: fig02_13.cpp
2 // Comparando inteiros utilizando instruções if, operadores relacionais
3 // e operadores de igualdade.
4 #include <iostream> // permite ao programa realizar entrada e saída
5
6 using std::cout; // o programa utiliza cout
7 using std::cin; // o programa utiliza cin
8 using std::endl; // o programa utiliza endl
9
10 // a função main inicia a execução do programa
11 int main()
12 {
13     int number1; // primeiro inteiro a comparar
14     int number2; // segundo inteiro a comparar
15
16     cout << "Enter two integers to compare: "; // solicita dados ao usuário
17     cin >> number1 >> number2; // lê dois inteiros fornecidos pelo usuário
18
19     if ( number1 == number2 )
20         cout << number1 << " == " << number2 << endl;
21
22     if ( number1 != number2 )
23         cout << number1 << " != " << number2 << endl;
24
25     if ( number1 < number2 )
26         cout << number1 << " < " << number2 << endl;
27
28     if ( number1 > number2 )
29         cout << number1 << " > " << number2 << endl;
30
31     if ( number1 <= number2 )
32         cout << number1 << " <= " << number2 << endl;
33
34     if ( number1 >= number2 )
35         cout << number1 << " >= " << number2 << endl;
36
37     return 0; // indica que o programa terminou com sucesso
38
39 } // fim da função main

```

Enter two integers to compare: **3 7**

3 != 7

3 < 7

3 <= 7

Enter two integers to compare: **22 12**

22 != 12

22 > 12

22 >= 12

Enter two integers to compare: **7 7**

7 == 7

7 <= 7

7 >= 7

Figura 2.13 Operadores de igualdade e operadores relacionais.

**Boa prática de programação 2.15***Coloque declarações us `using` imediatamente depois do `include#` que elas referenciam.*

As linhas 13–14

```
int number1; // primeiro inteiro a comparar
int number2; // segundo inteiro a comparar
```

declaram as variáveis utilizadas no programa. Lembre-se de que as variáveis podem ser declaradas em uma ou em múltiplas declarações.

O programa utiliza operações de extração de fluxo em cascata (linha 17) para inserir dois inteiros. Lembre-se de que podemos escrever `cin` (em vez de `std::cin`) por causa da linha 7. Primeiro um valor é lido para a variável `number1` e, então, um valor é lido para a variável `number2`.

A instrução `if` nas linhas 19–20

```
if ( number1 == number2 )
    cout << number1 << " == " << number2 << endl;
```

compara os valores de variáveis `number1` e `number2` para testá-los quanto à igualdade. Se os valores forem iguais, a instrução na linha 20 exibirá uma linha de texto para indicar que os números são iguais. Se as condições forem `true` em uma ou mais das instruções `if` que iniciam nas linhas 22, 25, 28, 31 e 34, a instrução correspondente do corpo exibirá uma linha apropriada de texto.

Note que cada instrução `if` na Figura 2.13 tem uma única instrução no seu corpo e que cada instrução do corpo é recuada. No Capítulo 4, mostramos como especificar instruções `if` com corpos de múltiplas instruções (incluindo as instruções de corpo em um par de chaves, `{ }`, criando o que é chamado de **instrução composta** ou **bloco**).

**Boa prática de programação 2.16***Recue a(s) instrução(ões) no corpo de uma instrução `if` para aprimorar a legibilidade.***Boa prática de programação 2.17***Para legibilidade não deve haver mais de uma instrução por linha em um programa.***Erro comum de programação 2.8**

Colocar um ponto-e-vírgula imediatamente depois do parêntese direito depois da condição em uma instrução `if` costuma ser um erro de lógica (embora não seja um erro de sintaxe). O ponto-e-vírgula faz com que o corpo da instrução `if` se esvazie, assim a instrução `if` não realiza nenhuma ação, independentemente de sua condição ser ou não verdadeira. Pior ainda, a instrução original do corpo da instrução `if` agora se tornaria uma instrução em sequência com a instrução `if` e sempre executaria, fazendo com que o programa produza resultados incorretos freqüentemente.

Observe o uso do espaço em branco na Figura 2.13. Recorde que os caracteres de espaço em branco como tabulações, nova linha e espaços normalmente são ignorados pelo compilador. Então, as instruções podem ser divididas em diversas linhas e espaçadas de acordo com as preferências do programador. É um erro de sintaxe dividir identificadores, strings (como "hello") e constantes (como o número 1000) em várias linhas.

**Erro comum de programação 2.9**

É um erro de sintaxe dividir um identificador inserindo caracteres de espaço em branco (por exemplo, escrevendo `main` como `ma in`).

**Boa prática de programação 2.18**

Uma instrução longa pode se estender por várias linhas. Se uma única instrução precisar ter sua linha quebrada, escolha pontos de quebra significativos, como depois de uma vírgula em uma lista separada por vírgulas ou depois de um operador em uma expressão extensa. Se uma instrução se estende por duas ou mais linhas, recue todas as linhas subseqüentes e alinhe à esquerda o grupo.

A Figura 2.14 mostra a precedência e a associatividade dos operadores introduzidos neste capítulo. Os operadores são mostrados de cima para baixo em ordem decrescente de precedência. Note que todos esses operadores, com a exceção do operador de atribuição `=`, se associam da esquerda para a direita. A adição associa da esquerda para a direita, então uma expressão como `x + y + z` é avaliada como se tivesse sido escrita `(x + y) + z`. O operador de atribuição `=` associa da direita para a esquerda, para que uma expressão como `x = y = 0` seja avaliada como se tivesse sido escrita `x = (y = 0)`, o que, como logo veremos, primeiro atribui 0 a `y` e então atribui o resultado dessa atribuição — 0 — a `x`.

Operadores	Associatividade	Tipo
()	da esquerda para a direita	parênteses
* / %	da esquerda para a direita	multiplicativo
+ -	da esquerda para a direita	aditivo
<< >>	da esquerda para a direita	inserção/extração de fluxo
< <= > >=	da esquerda para a direita	relacional
== !=	da esquerda para a direita	igualdade
=	da direita para a esquerda	atribuição

Figura 2.14 Precedência e associatividade dos operadores discutidos até agora.



Boa prática de programação 2.19

Consulte o gráfico de precedência e associatividade de operadores ao escrever expressões contendo muitos operadores. Confirme se os operadores na expressão são realizados na ordem em que você espera. Se não estiver certo quanto à ordem de avaliação em uma expressão complexa, divida a expressão em instruções menores ou utilize parênteses para forçar a ordem de avaliação, exatamente como faria em uma expressão algébrica. Certifique-se de observar que alguns operadores, como atribuição (=), associam da direita para a esquerda em vez de da esquerda para a direita.

2.8 Estudo de caso de engenharia de software: examinando o documento de requisitos de ATM (opcional)

Agora, iniciaremos nosso estudo de caso opcional orientado a objetos de projeto e implementação. As seções “Estudo de caso de engenharia de software” no final deste e dos vários próximos capítulos facilitarão seu entendimento da orientação a objeto. Desenvolveremos um software para um sistema de caixa eletrônico simples (*automated teller machine* – ATM), fornecendo uma experiência de projeto e implementação completa, cuidadosamente elaborada passo a passo. Nos capítulos 3–7, 9 e 13, realizaremos os vários passos do processo de um projeto orientado a objetos (*object-oriented design* – OOD) utilizando a UML e, ao mesmo tempo, relacionaremos esses passos aos conceitos orientados a objetos discutidos nos capítulos. O Apêndice G implementa o ATM utilizando as técnicas de programação orientada a objetos (*object-oriented programming* – OOP) em C++. Apresentamos a solução completa do estudo de caso. Isso não é um exercício; em vez disso, é uma experiência de aprendizagem completa que conclui com uma revisão detalhada do código C++ que implementa nosso projeto. Ele possibilitará que você se familiarize com os tipos de problemas substanciais encontrados na indústria e suas potenciais soluções.

Iniciamos nosso processo de projeto apresentando um **documento de requisitos** que especifica o propósito geral do sistema ATM e o que este deve fazer. Por todo o estudo de caso, nós nos referimos ao documento de requisitos para determinar precisamente a funcionalidade que o sistema deve incluir.

Documento de requisitos

Um banco local pretende instalar um novo caixa eletrônico (*automated teller machine* – ATM) para permitir que os usuários (isto é, os clientes do banco) realizem transações financeiras básicas (Figura 2.15). Cada usuário pode ter somente uma conta no banco. Os usuários do ATM devem ser capazes de visualizar seus saldos bancários, sacar dinheiro (isto é, retirar dinheiro de uma conta) e depositar fundos (isto é, colocar dinheiro em uma conta).

A interface com o usuário do caixa eletrônico contém os seguintes componentes de hardware:

- uma tela que exibe as mensagens para o usuário;
- um teclado que recebe a entrada numérica do usuário;
- um dispensador de cédulas que disponibiliza o dinheiro para o usuário e
- uma abertura para depósito que recebe os envelopes com o depósito do usuário.

O dispensador de cédulas é carregado diariamente com \$ 500 em notas de \$ 20. [Nota: Devido ao escopo limitado desse estudo de caso, certos elementos do ATM descritos aqui não simulam exatamente aqueles de um ATM real. Por exemplo, um ATM real em geral contém um dispositivo que lê o número da conta de um usuário a partir de um cartão ATM, enquanto esse ATM solicita que o usuário digite o número de uma conta no teclado. Normalmente, um ATM real também imprime um recibo no fim de uma sessão, mas toda a saída desse ATM aparece na tela.]

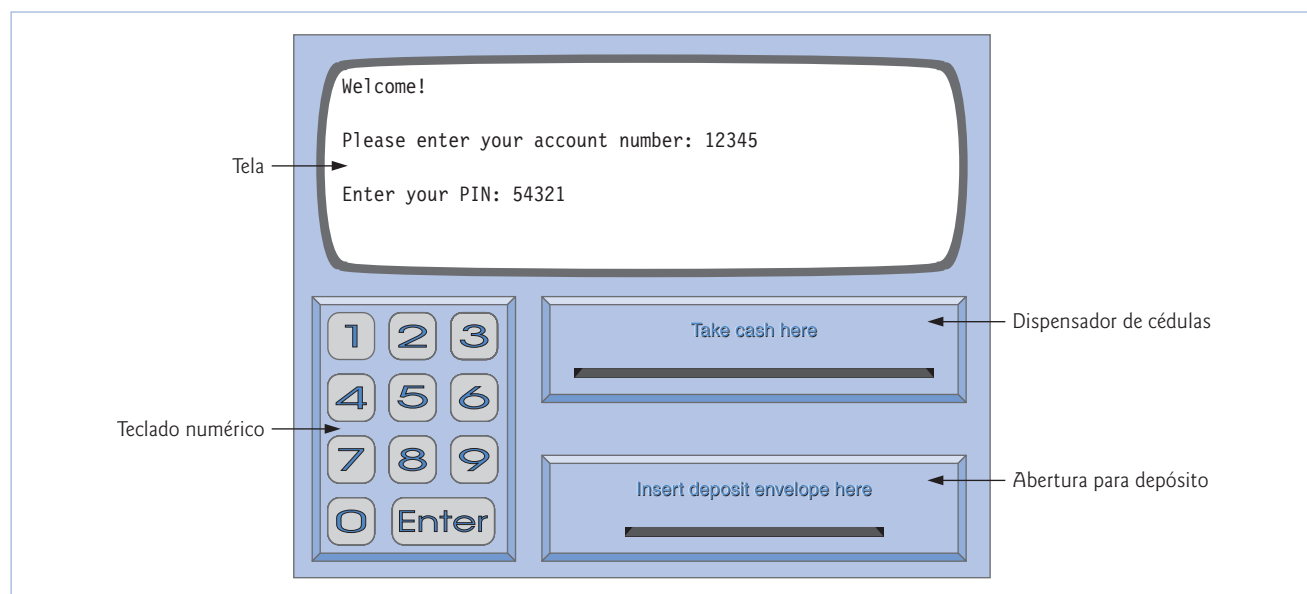


Figura 2.15 A interface com o usuário do caixa eletrônico.

O banco quer que você desenvolva um software para realizar as transações financeiras iniciadas pelos clientes do banco por meio do ATM. O banco integrará o software com o hardware do ATM em um momento posterior. O software deve encapsular a funcionalidade dos dispositivos de hardware (por exemplo, o dispensador de cédulas, a abertura para depósito) dentro dos componentes de software, mas ele próprio não precisa se preocupar com a maneira como esses dispositivos realizam suas tarefas. O hardware do ATM ainda não foi desenvolvido; assim, em vez de escrever seu software para ser executado no ATM, você deve desenvolver uma primeira versão do software para executar em um computador pessoal. Essa versão deve utilizar o monitor do computador para simular a tela do ATM e o teclado do computador para simular o teclado do ATM.

Uma sessão do ATM consiste na autenticação de um usuário (isto é, provar a identidade do usuário) com base em um número de conta e número pessoal de identificação (*personal identification number* – PIN) para então criar e executar as transações financeiras. Para autenticar um usuário e realizar transações, o ATM deve interagir com o banco de dados de informações de conta do banco. [Nota: Um banco de dados é uma coleção organizada de dados armazenados em um computador.] Para cada conta bancária, o banco de dados armazena um número de conta, um PIN e um saldo que indica a quantia de dinheiro na conta. [Nota: Por simplicidade, supomos que o banco planeja construir somente um ATM, portanto não precisamos nos preocupar com múltiplos ATMs acessando esse banco de dados ao mesmo tempo. Além disso, supomos que o banco não faz nenhuma alteração nas informações no banco de dados enquanto um usuário está acessando o ATM. Além disso, qualquer sistema de negócios como um ATM depara-se com questões de segurança razoavelmente complexas que estão bem além do escopo de um curso de ciência da computação no primeiro ou segundo semestre. Entretanto, fazemos a suposição simplificadora de que o banco confia no ATM para acesso e manipulação das informações no banco de dados sem medidas de segurança significativas.]

Ao acessar inicialmente o ATM, o usuário deve experimentar a seguinte sequência de eventos (mostrada na Figura 2.15):

1. A tela exibe uma mensagem de boas-vindas e solicita que o usuário insira o número da conta.
2. O usuário insere um número de conta de cinco dígitos utilizando o teclado numérico.
3. A tela solicita que o usuário insira o PIN associado com o número da conta especificada.
4. O usuário insere um PIN de cinco dígitos utilizando o teclado numérico.
5. Se o usuário inserir um número de conta válida e o PIN correto para essa conta, a tela exibe o menu principal (Figura 2.16). Se o usuário inserir um número inválido de conta ou um PIN incorreto, a tela exibe uma mensagem apropriada e então o ATM retorna ao *Passo 1* para reiniciar o processo de autenticação.

Depois que o ATM autentica o usuário, o menu principal (Figura 2.16) exibe uma opção numerada para cada um dos três tipos de transações: consulta de saldos (opção 1), retirada (opção 2) e depósito (opção 3). O menu principal também exibe uma opção que permite ao usuário sair do sistema (opção 4). O usuário então opta por realizar uma transação (inserindo 1, 2 ou 3) ou sair do sistema (inserindo 4). Se o usuário inserir uma opção inválida, a tela exibe uma mensagem de erro e, então, reexibe o menu principal.

Se o usuário inserir 1 para fazer uma consulta de saldos, a tela exibe o saldo da conta do usuário. Para fazer isso, o ATM deve recuperar o saldo a partir do banco de dados do banco.

As seguintes ações ocorrem quando o usuário insere 2 para fazer uma retirada:

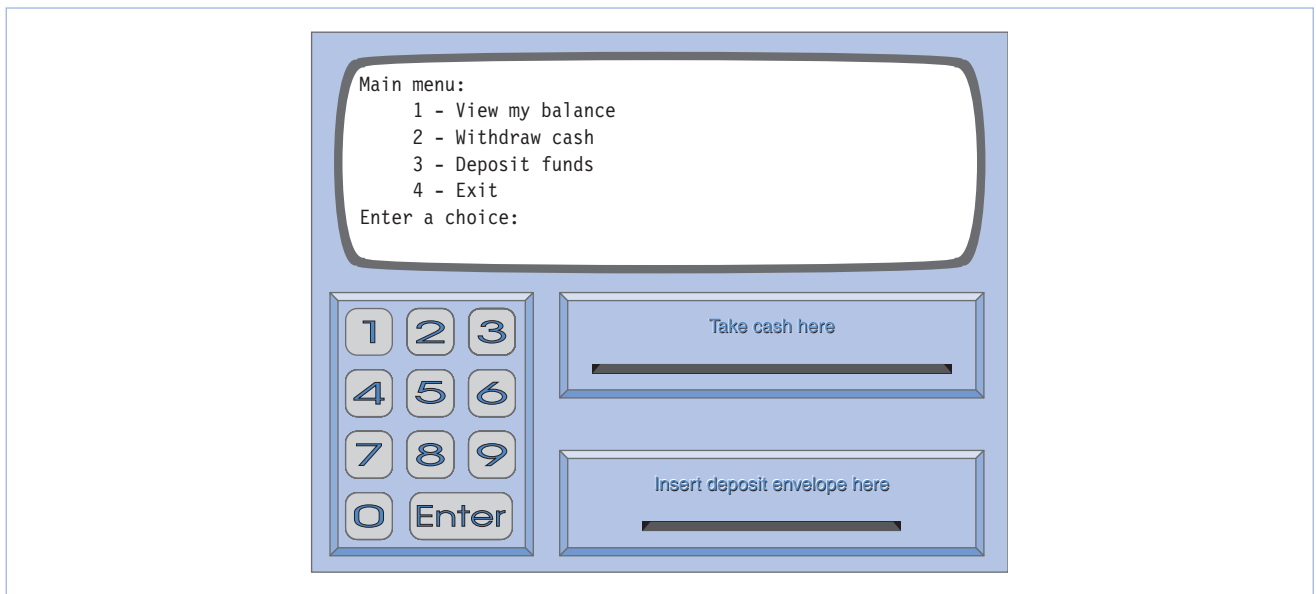


Figura 2.16 Menu principal do ATM.

6. A tela exibe um menu (mostrado na Figura 2.17) que contém quantias-padrão de saque: \$ 20 (opção 1), \$ 40 (opção 2), \$ 60 (opção 3), \$ 100 (opção 4) e \$ 200 (opção 5). O menu também contém uma opção para permitir que o usuário cancele a transação (opção 6).
7. O usuário insere uma seleção de menu (1–6) utilizando o teclado.
8. Se a quantia de saque escolhida for maior que o saldo da conta do usuário, a tela exibe uma mensagem declarando isso e solicitando que o usuário escolha uma quantia menor. O ATM então retorna ao *Passo 1*. Se a quantia de saque escolhida for menor ou igual ao saldo da conta do usuário (isto é, uma quantia de retirada aceitável), o ATM prossegue para o *Passo 4*. Se o usuário optar por cancelar a transação (opção 6), o ATM exibe o menu principal (Figura 2.16) e espera pela entrada do usuário.
9. Se o dispensador de cédulas contiver dinheiro suficiente para atender à solicitação, o ATM prossegue para o *Passo 5*. Do contrário, a tela exibe uma mensagem indicando o problema e solicitando que o usuário escolha uma quantia de saque menor. O ATM retorna então ao *Passo 1*.

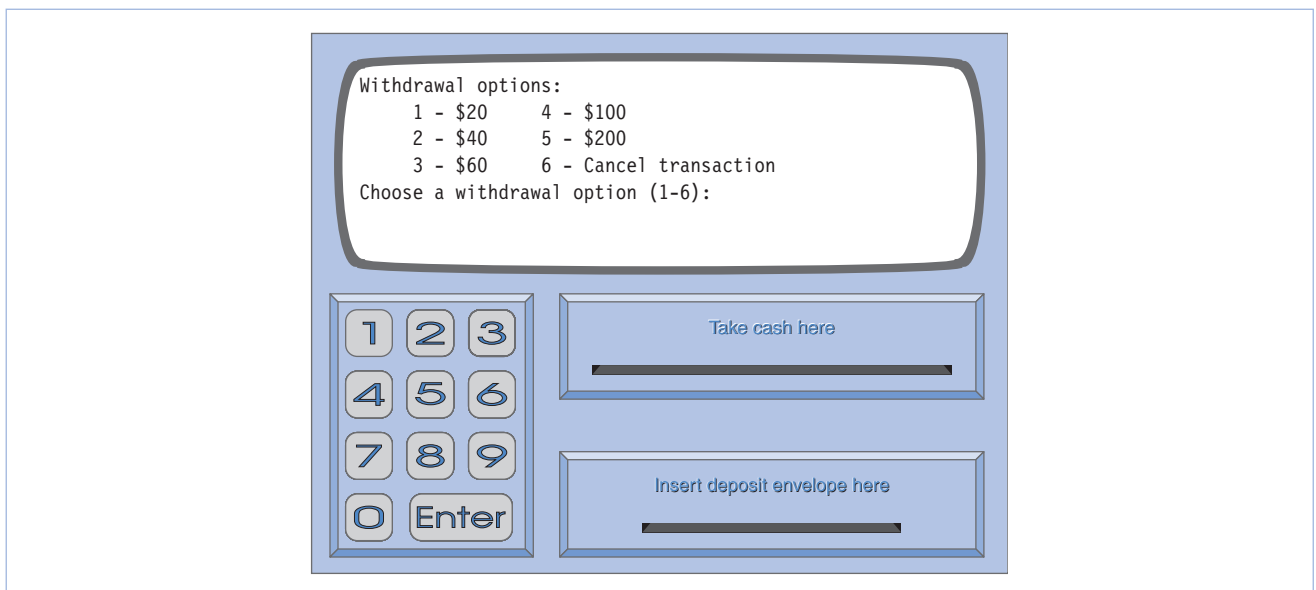


Figura 2.17 Menu de saque do ATM.

10. O ATM debita (isto é, subtrai) a quantia de saque do saldo da conta do usuário no banco de dados do banco.
11. O dispensador de cédulas entrega a quantia de dinheiro desejada para o usuário.
12. A tela exibe uma mensagem lembrando o usuário de pegar o dinheiro.

As seguintes ações ocorrem quando o usuário insere 3 (enquanto o menu principal é exibido) para efetuar um depósito:

13. A tela solicita que o usuário insira uma quantia de depósito ou que digite 0 (zero) para cancelar a transação.
14. O usuário insere uma quantia de depósito ou 0 utilizando o teclado numérico. [Nota: O teclado não contém um ponto de fração decimal nem um sinal de cifrão, portanto o usuário não pode digitar uma quantia monetária real (por exemplo, \$ 1,25). Em vez disso, o usuário deve inserir uma quantia de depósito como um número de centavos (por exemplo, 125). O ATM divide então esse número por 100 para obter um número que represente uma quantia monetária (por exemplo, $125 \div 100 = 1,25$).]
15. Se o usuário especificar uma quantia de depósito, o ATM prossegue para o *Passo 4*. Se o usuário optar por cancelar a transação (inserindo 0), o ATM exibe o menu principal (Figura 2.16) e espera pela entrada do usuário.
16. A tela exibe uma mensagem solicitando que o usuário insira um envelope de depósito na abertura de depósito.
17. Se a abertura de depósito receber um envelope de depósito dentro de dois minutos, o ATM credita (isto é, adiciona) a quantia de depósito ao saldo de conta de usuário no banco de dados do banco. [Nota: Essa quantia não permanece imediatamente disponível para saque. Primeiro o banco deve verificar fisicamente a quantia de dinheiro no envelope de depósito e quaisquer cheques no envelope devem ser transferidos do emissor do cheque para a conta do depositário. Quando qualquer um desses eventos ocorrer, o banco atualiza apropriadamente o saldo do usuário armazenado no seu banco de dados. Isso ocorre independentemente do sistema ATM.] Se a abertura para depósito não receber o envelope de um depósito dentro desse período, a tela exibe uma mensagem informando que o sistema cancelou a transação devido à inatividade. O ATM exibe então o menu principal e espera a entrada do usuário.

Depois que o sistema realiza com sucesso uma transação, ele deve reexibir o menu principal (Figura 2.16) para que o usuário possa realizar outras transações. Se o usuário escolher sair do sistema (opção 4), a tela exibe uma mensagem de agradecimento e então exibe a mensagem de boas-vindas para o próximo usuário.

Analizando o sistema ATM

A instrução anterior é um exemplo simplificado de um documento de requisitos. Em geral, esse documento é o resultado de um processo detalhado de **coleta de requisitos** que poderia incluir entrevistas com potenciais usuários do sistema e especialistas em campos relacionados ao sistema. Por exemplo, um analista de sistemas contratado para preparar um documento de requisitos para software de operações bancárias (por exemplo, o sistema ATM descrito aqui) poderia entrevistar especialistas financeiros para obter um melhor entendimento sobre o que o software deve fazer. O analista utilizaria as informações obtidas para compilar uma lista de **requisitos de sistema** a fim de orientar os projetistas de sistemas.

O processo de coleta de requisitos é uma tarefa-chave da primeira etapa do ciclo de vida de software. O **ciclo de vida do software** especifica as etapas pelas quais o software evolui desde o momento em que é inicialmente concebido até o momento em que é retirado de uso. Essas etapas em geral incluem: análise, projeto, implementação, teste e depuração, implantação, manutenção e retirada de uso. Há vários modelos de ciclo de vida de software, cada um com suas próprias preferências e especificações para o momento e a frequência com que os engenheiros de software devem realizar cada uma dessas etapas. **Modelos em cascata** realizam cada etapa uma vez em sucessão, enquanto **modelos iterativos** podem repetir uma ou mais etapas várias vezes por todo um ciclo de vida do produto.

A etapa da análise do ciclo de vida de software se concentra na definição do problema a ser solucionado. Ao projetar qualquer sistema, deve-se certamente *solucionar corretamente o problema*, mas, igualmente importante, deve-se *solucionar o problema correto*. Os analistas de sistemas coletam os requisitos que indicam o problema específico a ser solucionado. Nosso documento de requisitos descreve nosso sistema ATM em detalhes suficientes para que você não precise passar por uma extensa etapa de análise — isso já foi feito para você.

Para capturar o que um sistema proposto deve fazer, os desenvolvedores costumam empregar uma técnica conhecida como **modelagem de caso de uso**. Esse processo identifica os **casos de uso** do sistema, cada um dos quais representa uma capacidade diferente que o sistema fornece para seus clientes. Por exemplo, ATMs em geral têm vários casos de uso, como ‘Visualizar saldo de conta’, ‘Sacar dinheiro’, ‘Depositar fundos’, ‘Transferir fundos entre contas’ e ‘Comprar selos de postagem’. O sistema ATM simplificado que construímos nesse estudo de caso permite somente os três primeiros casos de uso (Figura 2.18).

Cada caso de uso descreve um cenário típico em que o usuário utiliza o sistema. Você já leu descrições dos casos de uso do sistema ATM no documento de requisitos; as listas de passos necessários para realizar cada tipo de transação (isto é, consulta de saldo, saque e depósito) na verdade descreveram os três casos de uso do nosso ATM — ‘Visualizar saldo em conta’, ‘Sacar dinheiro’ e ‘Depositar fundos’.

Diagramas de casos de uso

Agora introduzimos o primeiro de vários diagramas UML em nosso estudo de caso de ATM. Criamos um **diagrama de casos de uso** para modelar as interações entre os clientes de um sistema (neste estudo de caso, clientes do banco) e o sistema. O objetivo é mostrar os tipos de interações entre usuários e um sistema sem fornecer os detalhes — estes são fornecidos em outros diagramas UML (que

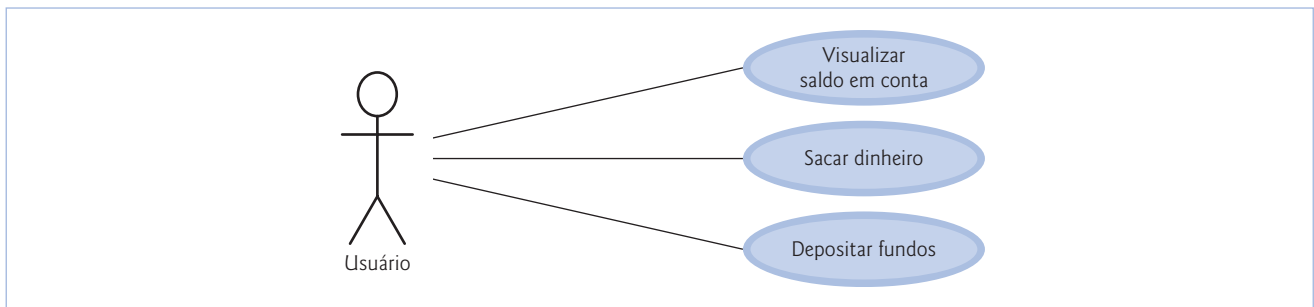


Figura 2.18 Diagrama de casos de uso para o sistema ATM da perspectiva do usuário.

apresentamos por todo o estudo de caso). Os diagramas de caso de uso costumam ser acompanhados por texto informal que descreve os casos de uso em mais detalhes — como o texto que aparece no documento de requisitos. Os diagramas de casos de uso são produzidos durante a etapa de análise do ciclo de vida do software. Em sistemas maiores, diagramas de casos de uso são ferramentas simples, mas indispensáveis, que ajudam projetistas de sistema a permanecer concentrados em atender às necessidades dos usuários.

A Figura 2.18 mostra o diagrama dos casos de uso para nosso sistema ATM. O boneco de palitos representa um **ator**, que define os papéis que uma entidade externa — como uma pessoa ou outro sistema — reproduz ao interagir com o sistema. Para nosso ATM, o ator é o Usuário que pode visualizar um saldo em conta, sacar dinheiro e depositar fundos na ATM. O Usuário não é uma pessoa real, mas em vez disso abrange os papéis que uma pessoa real — ao reproduzir a parte de um Usuário — pode reproduzir ao interagir com o ATM. Observe que um diagrama de casos de uso pode incluir múltiplos atores. Por exemplo, o diagrama de casos de uso para o sistema ATM de um banco real poderia também incluir um ator chamado Administrador que recarrega o dispensador de cédulas todos os dias.

Identificamos o ator em nosso sistema examinando o documento de requisitos, que declara ‘usuários de ATM devem ser capazes de ver o saldo em suas contas, sacar dinheiro e depositar fundos’. Portanto, o ator em cada um dos três casos de uso é o Usuário que interage com o ATM. Uma entidade externa — uma pessoa real — desempenha o papel do Usuário para realizar transações financeiras. A Figura 2.18 mostra um ator, cujo nome, Usuário, aparece abaixo do ator no diagrama. A UML modela cada caso de uso como uma oval conectada a um ator com uma linha sólida.

Os engenheiros de software (mais precisamente, os projetistas de sistemas) devem analisar o documento de requisitos ou um conjunto de casos de uso e projetar o sistema antes que os programadores o implementem em uma linguagem de programação particular. Durante a etapa de análise, os projetistas de sistemas se concentram no entendimento do documento de requisitos para produzir uma especificação de alto nível que descreve o que o sistema deve fazer. A saída da etapa de projeto — uma **especificação de projeto** — deve especificar claramente como o sistema deve ser construído a fim de satisfazer esses requisitos. Nas várias seções “Estudo de caso sobre engenharia de software” a seguir, realizamos os passos do processo de um projeto simples orientado a objetos (OOD) no sistema ATM para produzir uma especificação de projeto que contém uma coleção de diagramas da UML e texto de suporte. Lembre-se de que a UML é projetada para ser utilizada com qualquer processo OOD. Há muitos desses processos, dos quais o mais conhecido é o Rational Unified Process™ (RUP), desenvolvido pela Rational Software Corporation (agora uma divisão da IBM). O RUP é um processo rico concebido para projetar aplicativos com ‘poder industrial’. Para esse estudo de caso, apresentamos nosso próprio processo de projeto simplificado.

Projetando o sistema ATM

Agora, começamos a etapa de projeto do nosso sistema ATM. Um **sistema** é um conjunto de componentes que interage para resolver um problema. Por exemplo, para que o sistema ATM realize as tarefas projetadas, nosso sistema ATM tem uma interface com o usuário (Figura 2.15), contém um software que executa as transações financeiras e interage com um banco de dados das informações de conta bancária. A **estrutura do sistema** descreve os objetos do sistema e seus inter-relacionamentos. O **comportamento do sistema** descreve como o sistema muda à medida que seus objetos interagem entre si. Cada sistema apresenta tanto uma estrutura como um comportamento — os projetistas devem especificar ambos. Há vários tipos distintos de estruturas e comportamentos de sistema. Por exemplo, as interações entre objetos no sistema diferem daquelas entre o usuário e o sistema, contudo ambos constituem uma parte do comportamento de sistema.

A UML 2 especifica 13 tipos de diagramas para documentar os modelos dos sistemas. Cada um modela uma característica distinta da estrutura ou comportamento de um sistema — seis diagramas se relacionam com a estrutura do sistema; os sete remanescentes estão relacionados com o comportamento de sistema. Listamos aqui somente os seis tipos de diagramas utilizados em nosso estudo de caso — um desses (diagrama de classes) modela a estrutura do sistema — os cinco restantes modelam o comportamento do sistema. Fornecemos uma visão geral dos tipos de diagramas da UML remanescentes no Apêndice H, “UML 2: Tipos de diagrama adicionais”.

1. **Diagramas de casos de uso**, como o da Figura 2.18, modelam as interações entre um sistema e suas entidades externas (atores) em termos de casos de uso (capacidades do sistema, como ‘Visualizar saldo em conta’, ‘Sacar dinheiro’ e ‘Depositar fundos’).
2. **Diagramas de classes**, que você estudará na Seção 3.11, modelam as classes, ou os ‘blocos de construção’ utilizados em um sistema. Todo substantivo ou ‘coisa’ descrito no documento de requisitos é candidato a ser uma classe no sistema (por exemplo,

‘conta’, ‘teclado numérico’). Os diagramas de classes nos ajudam a especificar os relacionamentos estruturais entre partes do sistema. Por exemplo, o diagrama de classes do sistema ATM especificará que o ATM é fisicamente composto de uma tela, teclado, dispensador de cédulas e uma abertura para depósito.

3. **Diagramas de estados de máquina**, que você estudará na Seção 3.11, modelam como um objeto muda de estado. O **estado** de um objeto é indicado pelos valores de todos os atributos do objeto em um determinado momento. Quando um objeto muda de estado, esse objeto pode comportar-se diferentemente no sistema. Por exemplo, depois de validar o PIN de um usuário, o ATM passa do estado ‘usuário não autenticado’ para o estado ‘usuário autenticado’ quando o ATM permite ao usuário realizar transações financeiras (por exemplo, visualizar o saldo em conta, sacar dinheiro, depositar fundos).
4. **Diagramas de atividades**, que você também estudará na Seção 5.11, modelam a **atividade** de um objeto — o fluxo de trabalho do objeto (seqüência de eventos) durante a execução do programa. Um diagrama de atividades modela as ações que o objeto realiza e especifica a ordem em que ele realiza essas ações. Por exemplo, um diagrama de atividades mostra que o ATM deve obter o saldo em conta do usuário (a partir do banco de dados com as informações da conta bancária) antes de a tela poder exibir o saldo para o usuário.
5. **Diagramas de comunicação** (chamados **diagramas de colaboração** nas versões anteriores da UML) modelam as interações entre os objetos em um sistema, com ênfase em *quais* interações ocorrem. Você aprenderá na Seção 7.12 que esses diagramas mostram quais objetos devem interagir para realizar uma transação no ATM. Por exemplo, o ATM deve se comunicar com o banco de dados de informações da conta bancária para obter o saldo em uma conta.
6. **Diagramas de seqüência** também modelam as interações entre os objetos em um sistema, mas diferentemente dos diagramas de comunicação, eles enfatizam *quando* as interações ocorrem. Você aprenderá na Seção 7.12 que esses diagramas ajudam a mostrar a ordem em que as interações ocorrem ao executar uma transação financeira. Por exemplo, a tela solicita que o usuário insira uma quantia de saque antes de o dinheiro ser entregue.

Na Seção 3.11, continuamos a projetar nosso sistema ATM identificando as classes no documento de requisitos. Realizamos isso extraindo substantivos simples chave e substantivos compostos chave do documento de requisitos. Utilizando essas classes, desenvolveremos nosso primeiro rascunho do diagrama de classes que modela a estrutura do nosso sistema ATM.

Recursos na Internet e na Web

Os URLs a seguir fornecem as informações sobre o projeto orientado a objetos com a UML.

www-306.ibm.com/software/rational/uml/

Lista as perguntas feitas com frequência sobre a UML, fornecidas pela IBM Rational.

www.softdocwiz.com/Dictionary.htm

Hospeda o Unified Modeling Language Dictionary, que lista e define todos os termos utilizados na UML.

www-306.ibm.com/software/rational/offerings/design.html

Fornecer informações sobre o software da IBM Rational disponíveis para projetar sistemas. Fornece downloads gratuitos de versões para avaliação por 30 dias de vários produtos, como o IBM Rational Rose® XDE Developer.

www.embarcadero.com/products/describe/index.html

Fornecer uma licença gratuita para avaliação de 15 dias para a ferramenta de modelagem da Embarcadero Technologies® UML, o Describe™.

www.borland.com/together/index.html

Fornecer uma licença de 30 dias gratuita para download de uma versão de avaliação do Borland® Together® ControlCenter™ — uma ferramenta de desenvolvimento de software que suporta a UML.

www.ilogix.com/rhapsody/rhapsody.cfm

Fornecer uma licença de 30 dias gratuita para download de uma versão de avaliação do I-Logix Rhapsody® — um ambiente de desenvolvimento orientado ao modelo baseado em UML 2.

argouml.tigris.org

Contém informações e downloads do ArgoUML, uma ferramenta de UML gratuita e de código-fonte aberto.

www.objectsbydesign.com/books/booklist.html

Lista livros sobre a UML e projeto orientado a objetos.

www.objectsbydesign.com/tools/umltools_byCompany.html

Lista as ferramentas de software que utilizam a UML, como IBM Rational Rose, Embarcadero Describe, Sparx Systems Enterprise Architect, I-Logix Rhapsody e Gentleware Poseidon for UML.

www.oootips.org/ood-principles.html

Fornecer respostas à pergunta: ‘O que faz um bom projeto orientado a objetos?’

www.cetus-links.org/oo_uml.html

Introduz a UML e fornece links para inúmeros recursos da UML.

www.agilemodeling.com/essays/umlDiagrams.htm

Fornecer descrições e tutoriais detalhados sobre cada um dos 13 tipos de diagramas da UML.

Leituras recomendadas

Os livros a seguir fornecem informações sobre o projeto orientado a objetos com a UML.

Booch, G. *Object-oriented analysis and design with applications*, Third Edition. Boston: Addison-Wesley, 2004.

Eriksson, H., et al. *UML 2 toolkit*. New York: John Wiley, 2003.

Kruchten, P. *The rational unified process: an introduction*. Boston: Addison-Wesley, 2004.

Larman, C. *Applying UML and patterns: an introduction to object-oriented analysis and design*, Second Edition. Upper Saddle River, NJ: Prentice Hall, 2002.

Roques, P. *UML in practice: the art of modeling software systems demonstrated through worked examples and solutions*. New York: John Wiley, 2004.

Rosenberg, D., and K. Scott. *Applying use case driven object modeling with UML: an annotated e-commerce example*. Reading, MA: Addison-Wesley, 2001.

Rumbaugh, J., I. Jacobson and G. Booch. *The complete UML training course*. Upper Saddle River, NJ: Prentice Hall, 2000.

Rumbaugh, J., I. Jacobson and G. Booch. *The unified modeling language reference manual*. Reading, MA: Addison-Wesley, 1999.

Rumbaugh, J., I. Jacobson and G. Booch. *The unified software development process*. Reading, MA: Addison-Wesley, 1999.

Exercícios de revisão do estudo de caso de engenharia de software

- 2.1** Suponha que permitimos aos usuários do nosso sistema ATM transferir dinheiro entre duas contas bancárias. Modifique o diagrama de casos de uso da Figura 2.18 para refletir essa alteração.
- 2.2** _____ modelam as interações entre objetos em um sistema com ênfase em quando essas interações ocorrem.
- Diagramas de classes
 - Diagramas de seqüências
 - Diagramas de comunicação
 - Diagramas de atividades
- 2.3** Qual das opções a seguir lista as etapas de um ciclo de vida de software típico em uma ordem sequencial?
- projeto, análise, implementação, teste
 - projeto, análise, teste, implementação
 - análise, projeto, teste, implementação
 - análise, projeto, implementação, teste

Respostas aos exercícios de revisão do estudo de caso de engenharia de software

- 2.1** A Figura 2.19 contém um diagrama de casos de uso para uma versão modificada do nosso sistema ATM que também permite aos usuários transferir dinheiro entre contas.
- 2.2** b.
- 2.3** d.

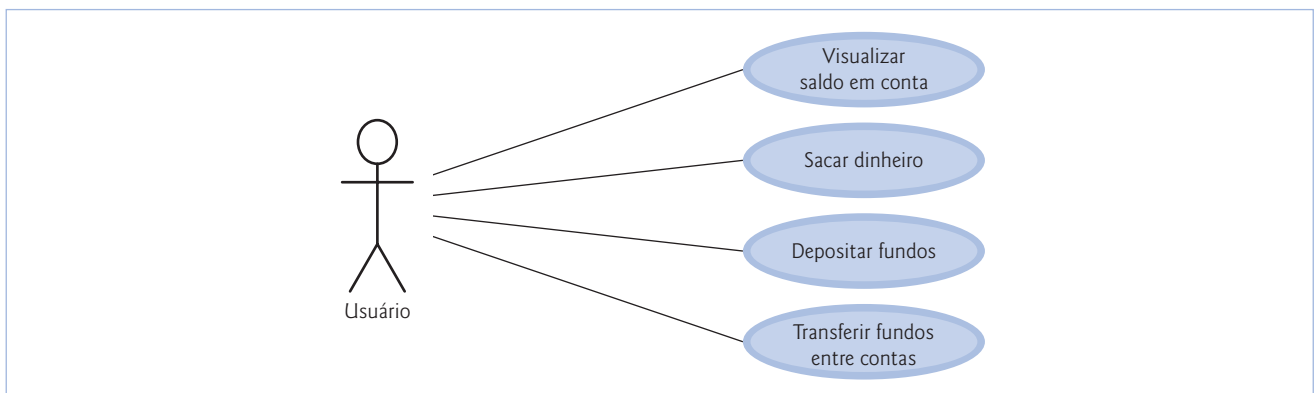


Figura 2.19 Diagrama de casos de uso para uma versão modificada do nosso sistema ATM que também permite aos usuários transferir dinheiro entre contas.

2.9 Síntese

Você aprendeu muitos recursos importantes do C++ neste capítulo, incluindo como exibir dados na tela, inserir dados a partir do teclado e declarar variáveis de tipos fundamentais. Em particular, você aprendeu a utilizar o objeto de fluxo de saída `cout` e o objeto de fluxo de entrada `cin` para construir programas interativos simples. Explicamos como as variáveis são armazenadas na memória e recuperadas a partir dela. Você também aprendeu a utilizar os operadores aritméticos para fazer cálculos. Discutimos a ordem em que o C++ aplica operadores (isto é, as regras de precedência de operadores), bem como a associatividade dos operadores. Você também aprendeu como a instrução `if` do C++ permite que um programa tome decisões. Por fim, introduzimos os operadores de igualdade e relacionais, que você utiliza para formar condições em instruções `if`.

Os aplicativos não orientados a objeto apresentados aqui introduziram os conceitos básicos de programação. Como você verá no Capítulo 3, os aplicativos C++ em geral contêm apenas algumas linhas de código na função `main` — essas instruções normalmente criam os objetos que fazem o trabalho do aplicativo e, então, os objetos ‘assumem a partir daí’. No Capítulo 3, você aprenderá a implementar suas próprias classes e a utilizar objetos dessas classes nos aplicativos.

Resumo

- Comentários de uma única linha iniciam com `//`. Os programadores inserem comentários para documentar programas e aprimorar sua legibilidade.
- Os comentários não fazem com que o computador realize qualquer ação quando o programa está em execução — eles são ignorados pelo compilador C++ e não fazem com que qualquer código-objeto de linguagem de máquina seja gerado.
- Uma diretiva de pré-processador inicia com `#` e é uma mensagem para o pré-processador C++. As diretivas de pré-processador são processadas pelo pré-processador antes de o programa ser compilado e não terminam com um ponto-e-vírgula como as instruções C++.
- A linha `#include <iostream>` instrui o pré-processador do C++ a incluir o conteúdo do arquivo de cabeçalho de fluxo de entrada/saída no programa. Esse arquivo contém as informações necessárias para compilar programas que utilizam `std::cin` e `std::cout` e operadores `<<` e `>>`.
- Os programadores utilizam espaço em branco (isto é, linhas em branco, caracteres de espaço em branco e caracteres de tabulação) para tornar os programas mais fáceis de ler. Os caracteres de espaço em branco são ignorados pelo compilador.
- Os programas C++ começam executando na função `main`, mesmo que `main` não apareça primeiro no programa.
- A palavra-chave `int` à esquerda de `main` indica que `main` ‘retorna’ um valor inteiro.
- Uma chave esquerda, `{`, deve iniciar o corpo de cada função. Uma chave direita correspondente, `}`, deve terminar o corpo de cada função.
- Uma string entre aspas duplas `"`, às vezes, referida como uma string de caracteres, mensagem ou literal string. Os caracteres de espaço em branco em strings não são ignorados pelo compilador.
- Cada instrução deve terminar com um ponto-e-vírgula (também conhecido como terminador de instrução).
- A saída e a entrada em C++ são feitas com fluxos de caracteres.
- O objeto de fluxo de saída `std::cout` — normalmente conectado à tela — é utilizado para gerar saída de dados. A saída de múltiplos itens de dados pode ser gerada concatenando operadores de inserção de fluxo (`<<`).
- O objeto de fluxo de entrada `std::cin` — normalmente conectado ao teclado — é utilizado para inserir dados. Múltiplos itens de dados podem ser inseridos concatenando-se os operadores de extração de fluxo (`>>`).
- Os objetos de fluxo `std::cout` e `std::cin` facilitam a interação entre o usuário e o computador. Como essa interação se assemelha a um diálogo, ela é freqüentemente chamada de computação conversacional ou computação interativa.
- A notação `std::cout` especifica que estamos utilizando um nome, nesse caso, `cout`, que pertence ao ‘namespace’ `std`.
- Quando uma barra invertida (isto é, um caractere de escape) é encontrada em uma string de caracteres, o próximo caractere é combinado com a barra invertida para formar uma sequência de escape.
- A sequência de escape `\n` significa nova linha. Ela faz com que o cursor (isto é, o indicador da posição de tela atual) se mova para o começo da próxima linha na tela.
- Uma mensagem que direciona o usuário para executar uma ação específica é conhecida como um prompt.
- A palavra-chave C++ `return` é um de vários meios de sair de uma função.
- Todas as variáveis em um programa C++ devem ser declaradas antes de ser utilizadas.
- Um nome de variável em C++ é qualquer identificador válido que não seja uma palavra-chave. Um identificador é uma série de caracteres consistindo em letras, dígitos e sublinhados (`_`). Os identificadores não podem iniciar com um dígito. Identificadores C++ podem ter qualquer comprimento; entretanto, alguns sistemas e/ou implementações C++ podem impor algumas restrições ao comprimento de identificadores.
- O C++ diferencia letras maiúsculas de minúsculas.
- A maioria dos cálculos é realizada em instruções de atribuição.

- Uma variável é uma posição na memória do computador onde um valor pode ser armazenado para utilização por um programa.
- As variáveis do tipo `int` armazenam valores inteiros, isto é, números inteiros como 7, -11, 0, 31.914.
- Cada variável armazenada na memória do computador tem um nome, um valor, um tipo e um tamanho.
- Sempre que um novo valor é colocado em uma posição da memória, o processo é destrutivo; isto é, o novo valor substitui o valor anterior nessa posição. O valor anterior é perdido.
- Quando um valor é lido a partir da memória, o processo é não destrutivo; isto é, uma cópia do valor é lida, deixando o valor original intacto na posição da memória.
- O manipulador de fluxo `std::endl` gera a saída de um caractere de nova linha e então ‘esvazia o buffer de saída’.
- O C++ avalia expressões aritméticas em uma sequência precisa determinada pelas regras de precedência e de associatividade de operadores.
- Parênteses podem ser utilizados para forçar que a ordem de avaliação ocorra em qualquer sequência desejada pelo programador.
- A divisão de inteiros (isto é, tanto o numerador como o denominador são inteiros) produz um quociente de inteiro. Qualquer parte fracionária na divisão de inteiros é truncada — não ocorre nenhum arredondamento.
- O operador módulo, `%`, fornece o resto de divisão de inteiros. O operador módulo pode ser utilizado somente com operandos inteiros.
- A instrução `if` permite que um programa tome uma decisão quando certa condição é satisfeita. O formato de uma instrução `if` é

```
if ( condição )
    instrução;
```

Se a condição for verdadeira, a instrução no corpo do `if` será executada. Se a condição não for satisfeita, isto é, a condição for falsa, a instrução de corpo não será executada.

- As condições em instruções `if` são comumente formadas com o uso de operadores de igualdade e operadores relacionais. O resultado do uso desses operadores é sempre o valor verdadeiro ou falso.
- A declaração

```
using std::cout;
```

é uma declaração `using` que elimina a necessidade de repetir o prefixo `std::`. Uma vez que incluímos essa declaração `using`, podemos escrever `cout` em vez de `std::cout` no restante de um programa.

Terminologia

`/* ... */` comentário (comentário no estilo C)
`//` comentário
`<` ‘é menor que’
`<=` ‘é menor que ou igual a’
`>` ‘é maior que’
`>=` ‘é maior que ou igual a’
arquivo de cabeçalho de fluxo de entrada/saída
`<iostream>`
associatividade da esquerda para a direita
associatividade de operadores
associatividade dos operadores
bloco
caractere de escape (`\`)
caractere de nova linha (`\n`)
`cin`, objeto
comentário (`//`)
condição
corpo de uma função
`cout`, objeto
cursor
decisão
declaração
diretiva de pré-processador
distinção de letras maiúsculas e minúsculas
divisão de inteiros

erro de compilação
erro de compilador
erro de lógica
erro de lógica não fatal
erro de sintaxe
erro em tempo de compilação
erro fatal
espaço em branco
fechar uma função
fluxo
função
gravação destrutiva
identificador
`if`, instrução
igualdade, operadores de
`!=` ‘é não igual a’
`==` ‘é igual a’
instrução
instrução composta
`int`, tipo de dados
inteiro (`int`)
leitura não destrutiva
lista separada por vírgulas
literal
literal string

`main`, função
manipulador de fluxo
memória
memória, posição
mensagem
objeto de fluxo de entrada padrão (`cin`)
objeto de fluxo de saída padrão (`cout`)
operações de inserção de fluxo de concatenação
operações de inserção de fluxo de encadeamento
operações de inserção de fluxo em cascata
operador
operador aritmético
operador binário
operador de atribuição (`=`)
operador de extração de fluxo (`>>`)
operador de inserção de fluxo (`<<`)
operador de multiplicação (`*`)
operador módulo (`%`)
operadores relacionais
operando
parênteses (`()`)
parênteses aninhados
parênteses redundantes
ponto-e-vírgula (`;`), terminador de instrução
precedência

programa autodocumentado	return, instrução	terminador de instrução (;)
prompt	seqüência de escape	tipo de dados
realizar uma ação	string	using, declaração
regras de precedência de operador	string de caractere	variável

Exercícios de revisão

2.1 Preencha as lacunas em cada uma das seguintes sentenças:

- Todo programa C++ inicia a execução na função _____.
- A _____ inicia o corpo de cada função e a _____ termina o corpo de cada função.
- Toda instrução em C++ termina com um(a) _____.
- A seqüência de escape `\n` representa o caractere _____, que faz com que o cursor se posicione no começo da próxima linha na tela.
- A instrução _____ é utilizada para tomar decisões.

2.2 Determine se cada uma das sentenças é *verdadeira* ou *falsa*. Se *falsa*, explique por quê. Suponha que a instrução `using std::cout;` é utilizada.

- Os comentários fazem com que o computador imprima o texto depois das `//` na tela quando o programa é executado.
- A seqüência de escape `\n`, quando tem sua saída gerada com `cout` e com o operador de inserção de fluxo, faz com que o cursor se posicione no começo da próxima linha na tela.
- Todas as variáveis devem ser declaradas antes de ser utilizadas.
- Todas as variáveis devem ser atribuídas a um tipo quando são declaradas.
- O C++ considera as variáveis `number` e `numbEr` idênticas.
- As declarações podem aparecer em quase qualquer lugar no corpo de uma função C++.
- O operador módulo (%) pode ser utilizado apenas com operandos inteiros.
- Os operadores aritméticos `*`, `/`, `%`, `+` e `-` têm o mesmo nível de precedência.
- Um programa C++ que imprime três linhas de saída deve conter três instruções que utilizam `cout` e o operador de inserção de fluxo.

2.3 Escreva uma única instrução C++ para realizar cada uma das seguintes tarefas (suponha que as declarações `using` não foram utilizadas):

- Declare as variáveis `c`, `thisIsAVariable`, `q76354` e `number` como sendo do tipo `String`.
- Solicite que o usuário insira um inteiro. Termine sua mensagem de solicitação com dois-pontos (:) seguido por um espaço e deixe o cursor posicionado depois do espaço.
- Leia um inteiro do usuário no teclado e armazene o valor inserido na variável do tipo inteiro `age`.
- Se a variável `number` não for igual a 7, imprima "The variable number is not equal to 7".
- Imprima a mensagem "This is a C++ program" em uma linha.
- Imprima a mensagem "This is a C++ program" em duas linhas. Termine a primeira linha com C++.
- Imprima a mensagem "This is a C++ program" com cada palavra em uma linha separada.
- Imprima a mensagem "This is a C++ program" com cada palavra separada da palavra seguinte por uma tabulação.

2.4 Escreva uma instrução (ou comentário) para realizar cada uma das seguintes sentenças (assuma que as declarações `using` foram utilizadas):

- Determine se um programa calcula o produto de três inteiros.
- Declare as variáveis `x`, `y`, `z` e `result` como tipo `int` (em instruções separadas).
- Peça para o usuário inserir três inteiros.
- Leia três inteiros a partir do teclado e armazene-os nas variáveis `x`, `y` e `z`.
- Compute o produto dos três inteiros contidos nas variáveis `x`, `y` e `z` e atribua o resultado à variável `result`.
- Imprima "The product is: " seguido pelo valor da variável `result`.
- Retorne um valor de `main` para indicar que o programa terminou com sucesso.

2.5 Utilizando as instruções que você escreveu no Exercício 2.4, escreva um programa completo que calcula e exhibe o produto de três inteiros. Onde apropriado, adicionar comentários ao código. [Nota: Você precisará escrever as declarações `using` necessárias.]

2.6 Identifique e corrija os erros em cada uma das seguintes instruções (suponha que a instrução `using std::cout;` é utilizada):

- `if ( c < 7 );`
`cout << "c is less than 7\n";`
- `if ( c => 7 )`
`cout << "c is equal to or greater than 7\n";`

Respostas dos exercícios de revisão

- 2.1** a) main. b) chave esquerda ({), chave direita (}). c) ponto-e-vírgula. d) nova linha. e) if.
- 2.2** a) Falsa. Os comentários não fazem com que qualquer ação seja realizada quando o programa é executado. Eles são utilizados para documentar programas e melhorar sua legibilidade.
 b) Verdadeira.
 c) Verdadeira.
 d) Verdadeira.
 e) Falsa. O C++ diferencia letras maiúsculas de minúsculas, então essas variáveis são únicas.
 f) Verdadeira.
 g) Verdadeira.
 h) Falsa. Os operadores *, / e % têm a mesma precedência, e os operadores + e - têm uma precedência menor.
 i) Falsa. Uma única instrução cout com múltiplas seqüências de escape \n podem imprimir várias linhas.
- 2.3** a) `int c, thisIsAVariable, q76354, number;`
 b) `std::cout << "Enter an integer: ";`
 c) `std::cin >> age;`
 d) `if ( number != 7 )`
 `std::cout << "The variable number is not equal to 7\n";`
 e) `std::cout << "This is a C++ program\n";`
 f) `std::cout << "This is a C++\nprogram\n";`
 g) `std::cout << "This\nis\na\nC++\nprogram\n";`
 h) `std::cout << "This\tis\ta\tC++\tprogram\n";`
- 2.4** a) // Calcula o produto de três inteiros
 b) `int x;`
 `int y;`
 `int z;`
 `int result;`
 c) `cout << "Enter three integers: ";`
 d) `cin >> x >> y >> z;`
 e) `result = x * y * z;`
 f) `cout << "The product is " << result << endl;`
 g) `return 0;`
- 2.5** (Ver programa abaixo)

```

1 // Calcula o produto de três inteiros
2 #include <iostream> // permite ao programa realizar entrada e saída
3
4 using std::cout; // o programa utiliza cout
5 using std::cin; // o programa utiliza cin
6 using std::endl; // o programa utiliza endl
7
8 // a função main inicia a execução do programa
9 int main()
10 {
11     int x; // primeiro inteiro a multiplicar
12     int y; // segundo inteiro a multiplicar
13     int z; // terceiro inteiro a multiplicar
14     int result; // o produto dos três inteiros
15
16     cout << "Enter three integers: "; // solicita dados ao usuário
17     cin >> x >> y >> z; // lê três inteiros de usuário
18     result = x * y * z; // multiplica os três inteiros; resultado de armazenamento
19     cout << "The product is " << result << endl; // imprime resultado; termina a linha
20
21     return 0; // indica que o programa executou com sucesso
22 } // fim da função main

```

- 2.6** a) Erro: Ponto-e-vírgula depois do parêntese direito da condição na instrução `if`.
Correção: Remova o ponto-e-vírgula depois do parêntese direito. [Nota: O resultado desse erro é que a instrução de saída será executada se a condição na instrução `if` for verdadeira.] O ponto-e-vírgula depois do parêntese direito é uma instrução nula (ou vazia) — uma instrução que não faz nada. Aprenderemos mais sobre a instrução nula no próximo capítulo.
- b) Erro: O operador relacional `=>`.
Correção: Mude `=>` para `>=` e você também pode querer mudar ‘igual a ou maior que’ para ‘maior que ou igual a’.

Exercícios

- 2.7** Discuta o significado de cada um dos seguintes objetos:
- `std::cin`
 - `std::cout`
- 2.8** Preencha as lacunas em cada uma das seguintes sentenças:
- _____ são utilizados para documentar um programa e aprimorar sua legibilidade.
 - O objeto utilizado para imprimir as informações na tela é _____.
 - Uma instrução C++ que toma uma decisão é _____.
 - A maioria de cálculos é geralmente feita por instruções _____.
 - O objeto _____ insere valores a partir do teclado.
- 2.9** Escreva uma única instrução C++ ou linha que realize cada uma das seguintes tarefas:
- Imprima a mensagem "Entre com dois números".
 - Atribua o produto de variáveis `b` e `c` para a variável `a`.
 - Declare que um programa realiza um cálculo de folha de pagamento (isto é, utilize texto que ajuda a documentar um programa).
 - Insira três valores de inteiro a partir do teclado nas variáveis de inteiro `a`, `b` e `c`.
- 2.10** Determine quais das seguintes sentenças são *verdadeiras* e quais são *falsas*. Se *falsa*, explique suas respostas.
- Operadores C++ são avaliados da esquerda para a direita.
 - Todos os seguintes nomes de variável são válidos: `_under_bar_`, `m928134`, `t5`, `j7`, `her_sales`, `his_account_total`, `a`, `b`, `c`, `z`, `z2`.
 - A instrução `cout << "a = 5;"` é um exemplo típico de uma instrução de atribuição.
 - Uma expressão aritmética em C++ válida sem parênteses é avaliada da esquerda para a direita.
 - Todos os seguintes nomes de variável são inválidos: `3g`, `87`, `67h2`, `h22`, `2h`.
- 2.11** Preencha as lacunas em cada uma das seguintes sentenças:
- Que operações aritméticas estão no mesmo nível de precedência que a multiplicação? _____.
 - Quando os parênteses são aninhados, qual conjunto de parênteses é avaliado primeiro em uma expressão aritmética? _____.
 - Uma posição na memória do computador que pode conter valores diferentes em várias vezes por toda a execução de um programa é chamada de _____.
- 2.12** O que é impresso, se algo for, quando cada uma das seguintes instruções C++ é executada? Se nada for impresso, então responda “nada”. Assuma `x = 2` e `y = 3`.
- `cout << x;`
 - `cout << x + x;`
 - `cout << "x=";`
 - `cout << "x = " << x;`
 - `cout << x + y << " = " << y + x;`
 - `z = x + y;`
 - `cin >> x >> y;`
 - `// cout << "x + y = " << x + y;`
 - `cout << "\n";`
- 2.13** Qual das instruções C++ a seguir contém variáveis cujos valores são substituídos?
- `cin >> b >> c >> d >> e >> f;`
 - `p = i + j + k + 7;`
 - `cout << "variables whose values are replaced";`
 - `cout << "a = 5";`
- 2.14** Dada a equação algébrica $y = ax^3 + 7$, qual das seguintes, se houver alguma, são instruções C++ corretas para essa equação?
- `y = a * x * x * x + 7;`
 - `y = a * x * x * ( x + 7 );`
 - `y = ( a * x ) * x * ( x + 7 );`
 - `y = ( a * x ) * x * x + 7;`

e) $y = a * (x * x * x) + 7;$

f) $y = a * x * (x * x + 7);$

2.15 Declare a ordem de avaliação dos operadores em cada uma das seguintes instruções C++ e mostre o valor de x depois que cada instrução é realizada.

a) $x = 7 + 3 * 6 / 2 - 1;$

b) $x = 2 \% 2 + 2 * 2 - 2 / 2;$

c) $x = (3 * 9 * (3 + (9 * 3 / (3))));$

2.16 Escreva um programa que solicita ao usuário inserir dois números, obtém os dois números do usuário e imprime a soma, produto, diferença e quociente dos dois números.

2.17 Escreva um programa que imprime os números 1 a 4 na mesma linha com cada par de números adjacentes separados por um espaço. Faça isso de várias maneiras:

a) Utilizando uma instrução com um operador de inserção de fluxo.

b) Utilizando uma instrução com quatro operadores de inserção de fluxo.

c) Utilizando quatro instruções.

2.18 Escreva um programa que pede para o usuário inserir dois inteiros, obtém os números do usuário, e então imprime o maior número seguido pelas palavras "é o maior". Se os números forem iguais, imprime a mensagem "Estes números são iguais".

2.19 Escreva um programa que insere três inteiros a partir do teclado e imprime a soma, a média, o produto, o menor e o maior desses números. O diálogo de tela deve se parecer com o seguinte:

```
Entre com três valores inteiros: 13 27 14
Soma: 54
Média: 18
Produto: 4914
O menor: 13
O maior: 27
```

2.20 Escreva um programa que lê o raio de um círculo como um inteiro e imprime o diâmetro, a circunferência e a área do círculo. Utilize o valor constante 3,14159 para π . Faça todos os cálculos em instruções de saída. [Nota: Neste capítulo, discutimos apenas as constantes e variáveis de inteiro. No Capítulo 4, discutimos números de ponto flutuante, isto é, valores que podem ter pontos de fração decimal.]

2.21 Escreva um programa que imprime uma caixa, uma oval, uma seta e um losango da seguinte maneira:

```
*****      ***      *      *
*           *      *      *      ***      *
*           *      *      *      *****   *
*           *      *      *      *           *
*           *      *      *      *           *
*           *      *      *      *           *
*           *      *      *      *           *
*           *      *      *      *           *
*           *      *      *      *           *
*****      ***      *           *
```

2.22 O que o seguinte código imprime?

```
cout << " *\n**\n***\n****\n*****" << endl;
```

2.23 Escreva um programa que lê cinco inteiros e determina e imprime o maior e o menor inteiro no grupo. Utilize somente as técnicas de programação que você aprendeu neste capítulo.

2.24 Escreva um programa que lê um inteiro e determina e imprime se ele é ímpar ou par. [Dica: Utilize o operador módulo. Um número par é um múltiplo de dois. Qualquer múltiplo de dois deixa um resto de zero quando dividido por 2.]

2.25 Escreva um programa que lê dois inteiros e determina e imprime se o primeiro é um múltiplo do segundo. [Dica: Utilize o operador módulo.]

2.26 Exiba o seguinte padrão de tabuleiro com oito instruções de saída e, em seguida, exiba o mesmo padrão utilizando o menor número de instruções possível.


```

* * * * *
 * * * * *
* * * * *
 * * * * *
* * * * *
 * * * * *
* * * * *
 * * * * *

```

- 2.27** Eis uma pequena antecipação do que está por vir. Neste capítulo, você aprendeu sobre inteiros e o tipo `int`. O C++ também pode representar letras maiúsculas, minúsculas e uma variedade considerável de símbolos especiais. O C++ utiliza inteiros pequenos internamente para representar cada caractere diferente. O conjunto de caracteres que um computador utiliza e das correspondentes representações na forma de inteiro desses caracteres é chamado **conjunto de caracteres** desse computador. Você pode imprimir um caractere colocando esse caractere entre aspas simples, como em

```
cout << 'A'; // imprime um A maiúsculo
```

Você pode imprimir o equivalente inteiro de um caractere utilizando `static_cast` da seguinte maneira:

```
cout << static_cast< int >( 'A' ); // imprime 'A' como um inteiro
```

Isso é chamado de operação de **coerção** (introduzimos formalmente as coerções no Capítulo 4). Quando a instrução precedente executar, ela imprimirá o valor 65 (em sistemas que utilizam o **conjunto de caracteres ASCII**). Escreva um programa que imprime o número inteiro equivalente de um caractere digitado no teclado. Teste seu programa várias vezes utilizando letras maiúsculas, minúsculas, dígitos e caracteres especiais (como \$).

- 2.28** Escreva um programa que insere um inteiro de cinco dígitos, separa o inteiro em seus dígitos individuais e imprime os dígitos separados entre si por três espaços cada. [Dica: Utilize operadores de divisão de inteiros e módulo.] Por exemplo, se o usuário digitar 42339, o programa deve imprimir:

```
4    2    3    3    9
```

- 2.29** Utilizando apenas as técnicas aprendidas neste capítulo, escreva um programa que calcula os quadrados e cubos dos inteiros de 0 a 10 e usa tabulações para imprimir as seguintes tabelas de valores, elegantemente formatadas:

inteiro	quadrado	cubo
0	0	0
1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216
7	49	343
8	64	512
9	81	729
10	100	1000



*Você verá uma coisa nova.
Duas coisas. E eu as chamo
Coisa Um e Coisa Dois.*
Dr. Theodor Seuss Geisel

*Nada pode ter valor sem ser um
objeto útil.*
Karl Marx

*Seus servidores públicos
prestam-lhe bons serviços.*
Adlai E. Stevenson

*Saber responder a quem fala,
Para responder a quem envia
uma mensagem.*
Amenemope

Introdução a classes e objetos

OBJETIVOS

Neste capítulo, você aprenderá:

- O que são classes, objetos, funções-membro e membros de dados.
- A definir uma classe e utilizá-la para criar um objeto.
- A definir funções-membro em uma classe para implementar os comportamentos da classe.
- A declarar membros de dados em uma classe para implementar os atributos da classe.
- A chamar uma função-membro de um objeto para fazê-la realizar sua tarefa.
- As diferenças entre membros de dados de uma classe e as variáveis locais de uma função.
- Como utilizar um construtor para assegurar que os dados de um objeto sejam inicializados quando o objeto for criado.
- Como projetar uma classe para separar sua interface de sua implementação e encorajar reutilização.

- 3.1** Introdução
- 3.2** Classes, objetos, funções-membro e membros de dados
- 3.3** Visão geral dos exemplos do capítulo
- 3.4** Definindo uma classe com uma função-membro
- 3.5** Definindo uma função-membro com um parâmetro
- 3.6** Membros de dados, funções *set* e funções *get*
- 3.7** Inicializando objetos com construtores
- 3.8** Colocando uma classe em um arquivo separado para reusabilidade
- 3.9** Separando a interface da implementação
- 3.10** Validando dados com funções *set*
- 3.11** Estudo de caso de engenharia de software: identificando as classes no documento de requisitos do ATM (opcional)
- 3.12** Síntese

Resumo | Terminologia | Exercícios de revisão | Respostas dos exercícios de revisão | Exercícios

3.1 Introdução

No Capítulo 2, você criou programas simples que exibiam mensagens para o usuário, informações obtidas do usuário, cálculos realizados e tomadas de decisão. Neste capítulo, você começará escrevendo programas que empregam os conceitos básicos da programação orientada a objetos que introduzimos na Seção 1.17. Um recurso comum de cada programa do Capítulo 2 era que todas as instruções que realizavam tarefas localizavam-se na função `main`. Em geral, os programas que você desenvolve neste livro consistirão na função `main` e em uma ou mais classes, cada uma contendo membros de dados e funções-membro. Se você se tornar parte de uma equipe de desenvolvimento na indústria, poderá trabalhar em sistemas de software com centenas, ou até milhares, de classes. Neste capítulo, desenvolvemos uma estrutura simples e bem projetada para organizar programas orientados a objetos em C++.

Primeiro, motivamos a noção de classes com um exemplo do mundo real. Em seguida, apresentamos uma sequência cuidadosamente elaborada de sete programas funcionais completos para demonstrar a criação e o uso de suas próprias classes. Esses exemplos iniciam nosso estudo de caso integrado no desenvolvimento de uma classe de livro de notas que os instrutores podem utilizar para manter um quadro de notas de testes dos alunos. Esse estudo de caso é aprimorado nos próximos capítulos, culminando com a versão apresentada no Capítulo 7, “Arrays e vetores”.

3.2 Classes, objetos, funções-membro e membros de dados

Vamos iniciar com uma analogia simples para ajudar a reforçar o entendimento da Seção 1.17 de classes e seu conteúdo. Suponha que você queira guiar um carro e fazê-lo andar mais rápido pisando no pedal acelerador. O que deve acontecer antes que você possa fazer isso? Bem, antes de você poder dirigir um carro, alguém tem de projetá-lo e construí-lo. Em geral, um carro inicia com os desenhos de engenharia, semelhantes às plantas utilizadas para projetar uma casa. Esses desenhos incluem o design de um pedal acelerador que o motorista usará para fazer o carro andar mais rápido. De certo modo, o pedal ‘oculta’ os complexos mecanismos que realmente fazem o carro ir mais rápido, assim como o pedal do freio ‘oculta’ os mecanismos que diminuem a velocidade do carro, o volante ‘oculta’ os mecanismos que mudam a direção do carro e assim por diante. Isso permite que pessoas com pouco ou nenhum conhecimento sobre como os carros são projetados possam dirigi-lo facilmente, simplesmente utilizando o acelerador, o freio, o volante, o mecanismo de troca de marcha e outras ‘interfaces’ simples e amigáveis ao usuário para os complexos mecanismos internos do carro.

Infelizmente, você não pode dirigir os desenhos de engenharia de um carro — antes que você possa dirigir um carro, ele precisa ser construído a partir dos desenhos de engenharia que o descrevem. Um carro pronto terá um acelerador real para fazer o carro andar mais rapidamente. Mas mesmo isso não é suficiente — o carro não acelerará sozinho, então o motorista deve pressionar o acelerador.

Agora vamos utilizar nosso exemplo do carro para introduzir os conceitos-chave de programação orientada a objeto desta seção. Realizar uma tarefa em um programa requer uma função (como `main`, conforme descrito no Capítulo 2). A função descreve os mecanismos que realmente realizam suas tarefas. A função oculta de seu usuário as tarefas complexas que ele realiza, assim como o pedal acelerador de um carro oculta do motorista os complexos mecanismos que fazem o carro andar mais rapidamente. Em C++, primeiro criamos uma unidade de programa chamada classe para abrigar uma função, assim como os desenhos de engenharia do carro abrigam o projeto de um pedal acelerador. Considerando a Seção 1.17, lembre-se de que uma função que pertence a uma classe é chamada de função-membro. Em uma classe, você fornece uma ou mais funções-membro que são projetadas para realizar as tarefas da classe. Por exemplo, uma classe que representa uma conta bancária poderia conter uma função-membro para depositar dinheiro na conta, outra para retirar dinheiro da conta e uma terceira para consultar o saldo atual da conta.

Assim como você não pode dirigir um desenho de engenharia de um carro, você não pode ‘dirigir’ uma classe. Assim como alguém tem de construir um carro a partir de desenhos de engenharia antes que o carro possa ser realmente dirigido, você deve criar um objeto de uma classe antes de fazer um programa realizar as tarefas que a classe descreve. Essa é uma razão pela qual C++ é conhecido como uma linguagem de programação orientada a objetos. Observe também que, assim como *muitos* carros podem ser construídos a partir do mesmo desenho de engenharia, *muitos* objetos podem ser construídos a partir da mesma classe.

Ao dirigir um carro, o ato de pressionar o acelerador envia uma mensagem para o carro realizar uma tarefa — isto é, fazer o carro andar mais rapidamente. De modo semelhante, você envia **mensagens** para um objeto — cada mensagem é conhecida como uma **chamada de função-membro** e diz para uma função-membro do objeto realizar sua tarefa. Isso é freqüentemente chamado de **solicitação de um serviço de um objeto**.

Até aqui utilizamos a analogia do carro para introduzir classes, objetos e funções-membro. Além das capacidades de um carro, ele também tem muitos atributos, como a cor, o número de portas, a quantidade de gasolina no tanque, a velocidade atual e o total de milhas percorrido (isto é, a leitura do odômetro). Como as capacidades do carro, esses atributos são representados como parte do projeto de um carro em seus diagramas de engenharia. Quando você dirige um carro, esses atributos estão sempre associados com o carro. Cada carro mantém seus próprios atributos. Por exemplo, cada carro sabe a quantidade de gasolina que há no seu tanque, mas não sabe quanto há no tanque de outros carros. De maneira semelhante, um objeto tem atributos que são portados com o objeto quando ele é utilizado em um programa. Esses atributos são especificados como parte da classe do objeto. Por exemplo, um objeto conta bancária tem um atributo saldo que representa a quantia de dinheiro na conta. Cada objeto conta bancária sabe o saldo da conta que ele representa, mas não sabe os saldos de outras contas no banco. Os atributos são especificados pelos membros de dados da classe.

3.3 Visão geral dos exemplos do capítulo

O restante deste capítulo apresenta sete exemplos simples que demonstram os conceitos que introduzimos no contexto da analogia do carro. Esses exemplos, resumidos a seguir, constroem incrementalmente uma classe `GradeBook` para demonstrar esses conceitos:

1. O primeiro exemplo apresenta uma classe `GradeBook` com uma função-membro que simplesmente exibe uma mensagem de boas-vindas quando é chamado. Então mostramos como criar um objeto dessa classe e chamar a função-membro para ele exibir a mensagem de boas-vindas.
2. O segundo exemplo modifica o primeiro, permitindo que a função-membro receba um nome do curso como um argumento. Então, a função-membro exibe o nome do curso como parte da mensagem de boas-vindas.
3. O terceiro exemplo mostra como armazenar o nome do curso em um objeto `GradeBook`. Para essa versão da classe, também mostramos como utilizar funções-membro para configurar o nome do curso no objeto e obter o nome do curso do objeto.
4. O quarto exemplo demonstra como os dados em um objeto `GradeBook` podem ser inicializados quando o objeto é criado — a inicialização é realizada por uma função-membro especial chamada construtor da classe. Esse exemplo também demonstra que cada objeto `GradeBook` mantém seu próprio membro de dados nome do curso.
5. O quinto exemplo modifica o quarto demonstrando como colocar a classe `GradeBook` em um arquivo separado para permitir a reusabilidade de software.
6. O sexto exemplo modifica o quinto demonstrando o bom princípio de engenharia de software de fazer uma separação entre a interface da classe e sua implementação. Isso torna a classe mais fácil de modificar sem afetar nenhum **cliente dos objetos da classe** — isto é, qualquer classe ou função que chame as funções-membro dos objetos da classe a partir de fora dos objetos.
7. O último exemplo aprimora a classe `GradeBook` introduzindo a validação de dados, que assegura que os dados em um objeto se adaptem a um formato particular ou estejam dentro de um intervalo de valores adequados. Por exemplo, um objeto `Date` exigiria um valor de mês no intervalo 1–12. Nesse exemplo de `GradeBook`, a função-membro que configura o nome do curso de um objeto `GradeBook` assegura que o nome do curso tenha 25 ou menos caracteres. Se não, a função-membro utiliza somente os primeiros 25 caracteres do nome do curso e exibe uma mensagem de advertência.

Observe que os exemplos de `GradeBook` neste capítulo realmente não processam nem armazenam notas. Começamos a processar as notas com a classe `GradeBook` no Capítulo 4 e as armazenamos em um objeto `GradeBook` no Capítulo 7, “Arrays e vetores”.

3.4 Definindo uma classe com uma função-membro

Iniciamos com um exemplo (Figura 3.1) que consiste na classe `GradeBook`, a qual representa um livro de notas que um instrutor pode utilizar para manter as notas que os alunos tiraram nas provas e uma função `main` (linhas 20–25) que cria um objeto `GradeBook`. Esse é o primeiro em uma série de exemplos graduados que preparam o caminho para uma classe `GradeBook` completamente funcional no Capítulo 7, “Arrays e vetores”. A função `main` utiliza esse objeto e sua função-membro para exibir na tela uma mensagem de boas-vindas ao instrutor para o programa de livro de notas.

Primeiro descrevemos como definir uma classe e uma função-membro. Depois explicamos como um objeto é criado e como chamar uma função-membro de um objeto. Os primeiros poucos exemplos contêm a função `main` e a classe `GradeBook` que ela utiliza no mesmo arquivo. Mais adiante no capítulo, introduzimos maneiras mais sofisticadas de estruturar programas para alcançar melhor engenharia de software.

```

1 // Figura 3.1: fig03_01.cpp
2 // Define a classe GradeBook com uma função-membro displayMessage;
3 // Cria um objeto GradeBook e chama sua função displayMessage.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 // Definição da classe GradeBook
9 class GradeBook
10 {
11 public:
12     // função que exibe uma mensagem de boas-vindas ao usuário do GradeBook
13     void displayMessage()
14     {
15         cout << "Welcome to the Grade Book!" << endl;
16     } // fim da função displayMessage
17 }; // fim da classe GradeBook
18
19 // a função main inicia a execução do programa
20 int main()
21 {
22     GradeBook myGradeBook; // cria um objeto GradeBook chamado myGradeBook
23     myGradeBook.displayMessage(); // chama a função displayMessage do objeto
24     return 0; // indica terminação bem-sucedida
25 } // fim de main

```

Welcome to the Grade Book!

Figura 3.1 Definindo a classe GradeBook com uma função-membro, criando um objeto GradeBook e chamando sua função-membro.

Classe *GradeBook*

Antes que a função `main` (linhas 20–25) possa criar um objeto da classe `GradeBook`, devemos informar ao compilador quais funções-membro e membros de dados pertencem à classe. Isso é conhecido como **definir uma classe**. A **definição da classe** `GradeBook` (linhas 9–17) contém uma função-membro chamada `displayMessage` (linhas 13–16) que exibe uma mensagem na tela (linha 15). Lembre-se de que uma classe é como uma planta arquitetônica — então precisamos fazer um objeto da classe `GradeBook` (linha 22) e chamar sua função-membro `displayMessage` (linha 23) para alcançar a linha 15 executar e exibir a mensagem de boas-vindas. Logo explicaremos as linhas 22–23 em detalhes.

A definição de classe inicia na linha 9 com a palavra-chave `class` seguida pelo nome da classe `GradeBook`. Por convenção, o nome de uma classe definida pelo usuário inicia com uma letra maiúscula e, por legibilidade, cada palavra subsequente no nome de classe inicia com uma letra maiúscula. Esse estilo de uso de letras maiúsculas e minúsculas é freqüentemente referido como **grafia camelo** (caixa alta e baixa), porque o padrão de letras maiúsculas e minúsculas se assemelha à silhueta de um camelo.

O **corpo** de cada classe é colocado entre um par de chaves esquerda e direita (`{ }`), como nas linhas 10 e 17. A definição de classe termina com um ponto-e-vírgula (linha 17).



Erro comum de programação 3.1

Esquecer do ponto-e-vírgula no fim de uma definição de classe é um erro de sintaxe.

Lembre-se de que a função `main` é sempre chamada automaticamente quando você executa um programa. A maioria das funções não é chamada automaticamente. Como logo verá, você deve chamar a função-membro `displayMessage` explicitamente para instruí-la a realizar sua tarefa.

A linha 11 contém o **rótulo especificador de acesso `public`**. A palavra-chave `public` é chamada de **especificador de acesso**. As linhas 13–16 definem a função-membro `displayMessage`. Essa função-membro aparece depois do especificador de acesso `public`: para indicar que a função está ‘disponível ao público’ — isto é, pode ser chamada por outras funções no programa e por funções-membro de outras classes. Os especificadores de acesso são sempre seguidos por dois-pontos (`:`). Para o restante do texto, quando nos referirmos ao especificador de acesso `public`, omitiremos o dois-pontos como fizemos nesta frase. A Seção 3.6 introduz um segundo especificador de acesso, `private` (novamente, omitimos os dois-pontos em nossas discussões, mas os incluímos em nossos programas).

Cada função em um programa realiza uma tarefa e pode retornar um valor quando ela completa sua tarefa — por exemplo, uma função poderia realizar um cálculo e, então, retornar o resultado desse cálculo. Quando você define uma função, deve especificar um **tipo de retorno** para indicar o tipo do valor retornado pela função quando ela completar sua tarefa. Na linha 13, a palavra-chave **void** à esquerda do nome de função `displayMessage` é o tipo de retorno da função. O tipo de retorno `void` indica que `displayMessage` realizará uma tarefa, mas não retornará (isto é, devolverá) dados para sua **função chamadora** (nesse exemplo, `main`, como veremos em breve) quando ela completar sua tarefa. (Na Figura 3.5, veremos um exemplo de uma função que retorna um valor.)

O nome da função-membro, `displayMessage`, segue o tipo de retorno. Por convenção, os nomes de função iniciam com a primeira letra minúscula e todas as palavras subsequentes no nome iniciam com uma letra maiúscula. Os parênteses depois do nome de função-membro indicam que isso é uma função. Um conjunto vazio de parênteses, como mostrado na linha 13, indica que essa função-membro não requer dados adicionais para realizar sua tarefa. Você verá um exemplo de uma função-membro que requer dados adicionais na Seção 3.5. A linha 13 é comumente referida como o **cabeçalho de função**. O corpo de cada função é delimitado pelas chaves esquerda e direita (`{` e `}`), como nas linhas 14 e 16.

O corpo de uma função contém instruções que realizam a tarefa da função. Nesse caso, a função-membro `displayMessage` contém uma instrução (linha 15) que exibe a mensagem "Welcome to the Grade Book!". Depois que essa instrução executar, a função terá concluído sua tarefa.



Erro comum de programação 3.2

Retornar um valor de uma função cujo tipo de retorno foi declarado como void é um erro de compilação.



Erro comum de programação 3.3

Definir uma função dentro de outra função é um erro de sintaxe.

Testando a classe **GradeBook**

Em seguida, gostaríamos de utilizar a classe `GradeBook` em um programa. Como você aprendeu no Capítulo 2, a função `main` começa a execução de cada programa. As linhas 20–25 da Figura 3.1 contêm a função `main` que controlará a execução do nosso programa.

Nesse programa, gostaríamos de chamar a função-membro `displayMessage` da classe `GradeBook` para exibir a mensagem de boas-vindas. Em geral, você não pode chamar uma função-membro de uma classe até criar um objeto dessa classe. (Como você aprenderá na Seção 10.7, as funções-membro `static` são uma exceção.) A linha 22 cria um objeto da classe `GradeBook` chamada `myGradeBook`. Observe que o tipo da variável é `GradeBook` — a classe que definimos nas linhas 9–17. Quando declaramos variáveis do tipo `int`, como fizemos no Capítulo 2, o compilador sabe o que é `int` — é um tipo fundamental. Quando escrevemos a linha 22, porém, o compilador não sabe automaticamente de que tipo é `GradeBook` — ele é um **tipo definido pelo usuário**. Portanto, devemos informar ao compilador de que tipo `GradeBook` é, incluindo a definição de classe, como fizemos nas linhas 9–17. Se omitíssemos essas linhas, o compilador emitiria uma mensagem de erro (como `'GradeBook': undeclared identifier` no Microsoft Visual C++ .NET ou `'GradeBook': undeclared` no GNU C++). Cada nova classe que você cria torna-se um novo tipo que pode ser utilizado para criar objetos. Os programadores podem definir novos tipos de classe conforme necessário; essa é uma razão pela qual o C++ é conhecido como uma **linguagem extensível**.

A linha 23 chama a função-membro `displayMessage` (definida nas linhas 13–16) utilizando a variável `myGradeBook` seguida pelo **operador ponto** (`.`), o nome de função `displayMessage` e um conjunto vazio de parênteses. Essa chamada faz com que a função `displayMessage` realize sua tarefa. No começo da linha 23, `'myGradeBook.'` indica que `main` deve utilizar o objeto `GradeBook` que foi criado na linha 22. Os parênteses vazios na linha 13 indicam que a função-membro `displayMessage` não requer dados adicionais para realizar sua tarefa. (Na Seção 3.5, você verá como passar dados para uma função.) Quando `displayMessage` completa sua tarefa, a função `main` continua executando na linha 24, o que indica que `main` realizou suas tarefas com sucesso. Esse é o fim de `main`, portanto o programa termina.

Diagrama de classes UML para classe **GradeBook**

A partir da Seção 1.17, lembre-se de que a UML é uma linguagem gráfica utilizada por programadores para representar sistemas orientados a objetos de maneira padronizada. Na UML, cada classe é modelada em um diagrama de classes como um retângulo com três compartimentos. A Figura 3.2 apresenta um **diagrama de classes UML** para a classe `GradeBook` da Figura 3.1. O compartimento superior contém o nome da classe centralizado horizontalmente e no tipo negrito. O compartimento do meio contém os atributos da classe que correspondem a membros de dados em C++. Na Figura 3.2, o compartimento do meio está vazio porque a versão da classe `GradeBook` na Figura 3.1 não tem nenhum atributo. (A Seção 3.6 apresenta uma versão da classe `GradeBook` que realmente tem um atributo.) O compartimento inferior contém as operações da classe que correspondem às funções-membro em C++. A UML modela operações listando o nome da operação seguido por um conjunto de parênteses. A classe `GradeBook` tem apenas uma função-membro, `displayMessage`, então o compartimento inferior da Figura 3.2 lista uma operação com esse nome. A função-membro `displayMessage` não requer informações adicionais para realizar suas tarefas, então os parênteses depois de `displayMessage` no diagrama de classes estão vazios, da mesma forma que estão no cabeçalho da função-membro na linha 13 da Figura 3.1. O sinal de adição (+) em frente do nome de operação indica que `displayMessage` é uma operação pública na UML (isto é, uma função-membro `public` em C++). Utilizamos freqüentemente diagramas de classes UML para resumir os atributos e operações de classe.

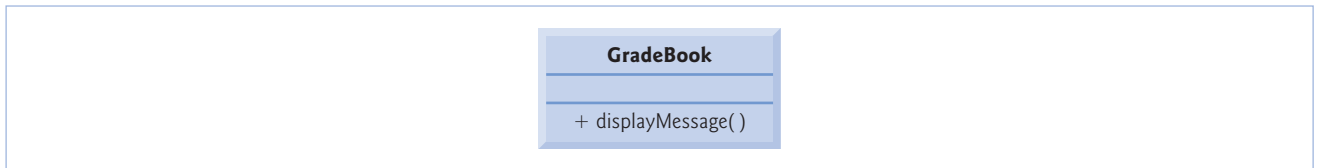


Figura 3.2 Diagrama de classes UML indicando que a classe `GradeBook` tem uma operação `displayMessage` pública.

3.5 Definindo uma função-membro com um parâmetro

Em nossa analogia do carro da Seção 3.2, mencionamos que pressionar o acelerador de um carro envia uma mensagem para ele realizar uma tarefa — fazer o carro andar mais rapidamente. Mas quanto o carro deve acelerar? Como você sabe, quanto mais pressionar o pedal, mais o carro irá acelerar. Então a mensagem para o carro inclui tanto a tarefa a ser realizada como informações adicionais que ajudam o carro a executar essa tarefa. Essas informações adicionais são conhecidas como um **parâmetro** — o valor do parâmetro ajuda o carro a determinar com que rapidez acelerar. De maneira semelhante, uma função-membro pode exigir um ou mais parâmetros que representam dados adicionais de que ela precisa para realizar sua tarefa. Uma chamada de função fornece valores — chamados de **argumentos** — para cada um dos parâmetros da função. Por exemplo, para fazer um depósito em uma conta bancária, suponha que uma função-membro `deposit` de uma classe `Account` especifique um parâmetro que representa a quantia do depósito. Quando a função-membro `deposit` é chamada, o valor de um argumento que representa a quantia de depósito é copiado para o parâmetro da função-membro. A função-membro então adiciona essa quantia ao saldo da conta.

Definindo e testando a classe `GradeBook`

Nosso próximo exemplo (Figura 3.3) redefine a classe `GradeBook` (linhas 14–23) com uma função-membro `displayMessage` (linhas 18–22) que exibe o nome do curso como parte da mensagem de boas-vindas. A nova função-membro `displayMessage` requer um parâmetro (`courseName` na linha 18) que representa o nome do curso a ser enviado para a saída.

Antes de discutir os novos recursos da classe `GradeBook`, vejamos como a nova classe é utilizada em `main` (linhas 26–40). A linha 28 cria uma variável de tipo `string` chamada `nameOfCourse` que será utilizada para armazenar o nome do curso inserido pelo usuário. Uma variável do tipo `string` representa uma string de caracteres como "CS101 Introduction to C++ Programming". Uma `string` é, de fato, um objeto da classe `string` da C++ Standard Library. Essa classe é definida no **arquivo de cabeçalho `<string>`** e o nome `string`, como `cout`, pertence ao namespace `std`. Para permitir que a linha 28 compile, a linha 9 inclui o arquivo de cabeçalho `<string>`. Observe que a declaração `using` na linha 10 permite simplesmente escrever `string` na linha 28 em vez de `std::string`. Por enquanto, você pode pensar em variáveis `string` como variáveis de outros tipos como `int`. Você aprenderá as capacidades adicionais de `string` na Seção 3.10.

A linha 29 cria um objeto da classe `GradeBook` chamado `myGradeBook`. A linha 32 exibe um prompt que pede para usuário inserir o nome de um curso. A linha 33 lê o nome do usuário e o atribui à variável `nameOfCourse`, utilizando a função de biblioteca **`getline`** para realizar a entrada. Antes de explicarmos essa linha de código, vamos explicar por que simplesmente não podemos escrever

```
cin >> nameOfCourse;
```

para obter o nome do curso. Em nosso exemplo de execução de programa, utilizamos o nome do curso "CS101 Introduction to C++ Programming", que contém múltiplas palavras. (Lembre-se de que destacamos a entrada fornecida pelo usuário em negrito.) Quando `cin` é utilizado com o operador de extração de fluxo, ele lê os caracteres até o primeiro caractere de espaço em branco ser alcançado. Portanto, somente "CS101" seria lido pela instrução precedente. O restante do nome do curso teria de ser lido por operações de entrada subsequentes.

Nesse exemplo, gostaríamos que o usuário digitasse o nome completo do curso e pressionasse *Enter* para enviá-lo para o programa e gostaríamos de armazenar o nome inteiro de curso na variável `string nameOfCourse`. A chamada de função `getline( cin, nameOfCourse )` na linha 33 lê caracteres (incluindo os caracteres de espaço em branco que separam as palavras na entrada) do objeto de fluxo de entrada-padrão `cin` (isto é, o teclado) até o caractere nova linha ser encontrado, coloca os caracteres na variável `string nameOfCourse` e descarta o caractere de nova linha. Observe que, quando você pressionar *Enter* ao digitar a entrada de programa, um caractere de nova linha é inserido no fluxo de entrada. Note também que o arquivo de cabeçalho `<string>` deve ser incluído no programa para utilizar a função `getline` e que o nome `getline` pertence ao namespace `std`.

A linha 38 chama a função-membro `displayMessage` do `myGradeBook`. A variável `nameOfCourse` em parênteses é o argumento que é passado para a função-membro `displayMessage` para que ela possa realizar sua tarefa. O valor da variável `nameOfCourse` em `main` torna-se o valor do parâmetro `courseName` da função-membro `displayMessage` na linha 18. Ao executar esse programa, note que a saída da função-membro `displayMessage` é gerada como parte da mensagem de boas-vindas do nome do curso que você digitou (em nosso exemplo de execução, CS101 Introduction to C++ Programming).

Mais sobre argumentos e parâmetros

Para especificar que uma função requer dados para realizar sua tarefa, você coloca informações adicionais na **lista de parâmetros** da função que está localizada entre os parênteses depois do nome de função. A lista de parâmetros pode conter qualquer número de parâ-

```

1 // Figura 3.3: fig03_03.cpp
2 // Define a classe GradeBook com uma função-membro que aceita um parâmetro;
3 // Cria um objeto GradeBook e chama sua função-membro displayMessage.
4 #include <iostream>
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include <string> // o programa utiliza classe de string padrão C++
10 using std::string;
11 using std::getline;
12
13 // definição da classe GradeBook
14 class GradeBook
15 {
16 public:
17     // função que exibe uma mensagem de boas-vindas ao usuário do GradeBook
18     void displayMessage( string courseName )
19     {
20         cout << "Welcome to the grade book for\n" << courseName << "!"
21         << endl;
22     } // fim da função displayMessage
23 }; // fim da classe GradeBook
24
25 // a função main inicia a execução do programa
26 int main()
27 {
28     string nameOfCourse; // strings de caracteres para armazenar o nome do curso
29     GradeBook myGradeBook; // cria um objeto GradeBook chamado myGradeBook
30
31     // prompt para entrada do nome do curso
32     cout << "Please enter the course name:" << endl;
33     getline( cin, nameOfCourse ); // lê o nome de um curso com espaços em branco
34     cout << endl; // gera saída de uma linha em branco
35
36     // chama a função displayMessage de myGradeBook
37     // e passa nameOfCourse como um argumento
38     myGradeBook.displayMessage( nameOfCourse );
39     return 0; // indica terminação bem-sucedida
40 } // fim de main

```

Please enter the course name:

CS101 Introduction to C++ Programming

Welcome to the grade book for

CS101 Introduction to C++ Programming!

Figura 3.3 Definindo a classe GradeBook com uma função-membro que aceita um parâmetro.

metros, incluindo nenhum (representado por parênteses vazios como na Figura 3.1, linha 13) para indicar que uma função não requer nenhum parâmetro. A lista de parâmetros da função-membro `displayMessage` (Figura 3.3, linha 18) declara que a função requer um parâmetro. Todo parâmetro deve especificar um tipo e um identificador. Nesse caso, o tipo `string` e o identificador `courseName` indicam que a função-membro `displayMessage` requer uma `string` para realizar sua tarefa. O corpo da função-membro utiliza o parâmetro `courseName` para acessar o valor que é passado para a função na chamada de função (linha 38 em `main`). As linhas 20–21 exibem o valor do parâmetro `courseName` como parte da mensagem de boas-vindas. Observe que o nome da variável de parâmetro (linha 18) pode ter ou não o mesmo nome da variável de argumento (linha 38) — você aprenderá por que isso acontece no Capítulo 6, “Funções e uma introdução à recursão”.

Uma função pode especificar múltiplos parâmetros separando cada parâmetro do próximo com uma vírgula (veremos um exemplo nas figuras 6.4–6.5). O número e a ordem de argumentos em uma chamada de função devem corresponder ao número e à ordem de parâmetros na lista de parâmetros do cabeçalho da função-membro chamada. Além disso, os tipos de argumento na chamada de função devem corresponder aos tipos dos parâmetros correspondentes no cabeçalho de função. (Como você aprenderá nos capítulos subsequentes, o tipo de um argumento e o tipo do seu parâmetro correspondente nem sempre precisam ser idênticos, mas devem ser ‘consistentes’.) Em nosso exemplo, o argumento `string` na chamada de função (isto é, `nameOfCourse`) corresponde exatamente ao parâmetro `string` na definição da função-membro (isto é, `courseName`).



Erro comum de programação 3.4

Colocar um ponto-e-vírgula após o parêntese direito que envolve a lista de parâmetros de uma definição de função é um erro de sintaxe.



Erro comum de programação 3.5

Definir um parâmetro de função novamente como uma variável local na função é um erro de compilação.



Boa prática de programação 3.1

Para evitar ambigüidade, não utilize os mesmos nomes para os argumentos passados para uma função e os parâmetros correspondentes na definição de função.



Boa prática de programação 3.2

Escolher nomes significativos para funções e parâmetros torna os programas mais legíveis e ajuda a evitar utilização excessiva de comentários.

Diagrama da classe UML atualizado para a classe **GradeBook**

O diagrama da classe UML da Figura 3.4 modela a classe `GradeBook` da Figura 3.3. Como a classe `GradeBook` definida na Figura 3.1, essa classe `GradeBook` contém a função-membro `public displayMessage`. Entretanto, essa versão de `displayMessage` tem um parâmetro. A UML modela um parâmetro listando o nome de parâmetro, seguido por dois-pontos e o tipo de parâmetro entre os parênteses que se seguem ao nome de operação. A UML tem seus próprios tipos de dados semelhantes aos do C++. A UML é independente de linguagem — é utilizada com muitas linguagens de programação diferentes — então sua terminologia não corresponde exatamente à do C++. Por exemplo, o tipo UML `String` corresponde ao tipo C++ `string`. A função-membro `displayMessage` da classe `GradeBook` (Figura 3.3; linhas 18–22) tem um parâmetro `string` chamado `courseName`, então a Figura 3.4 lista `courseName : String` entre os parênteses que se seguem ao nome de operação `displayMessage`. Observe que essa versão da classe `GradeBook` ainda não tem nenhum membro de dados.

3.6 Membros de dados, funções *set* e funções *get*

No Capítulo 2, declaramos todas as variáveis de um programa em sua função `main`. As variáveis declaradas em um corpo da definição de função são conhecidas como **variáveis locais** e só podem ser utilizadas a partir da linha de sua declaração na função até a chave direita de fechamento imediatamente seguinte (`}`) da definição de função. Uma variável local deve ser declarada antes de poder ser utilizada em uma função. Uma variável local não pode ser acessada fora da função em que é declarada. Quando uma função termina, os valores de suas variáveis locais são perdidos. (Você verá uma exceção a isso no Capítulo 6 quando discutirmos as variáveis locais `static`.) A partir da Seção 3.2, lembre-se de que um objeto tem atributos que são portados com esse objeto quando ele é utilizado em um programa. Esses atributos existem por toda a vida do objeto.

Uma classe normalmente consiste em uma ou mais funções-membro que manipulam os atributos que pertencem a um objeto particular da classe. Os atributos são representados como variáveis em uma definição de classe. Essas variáveis são chamadas de **membros**

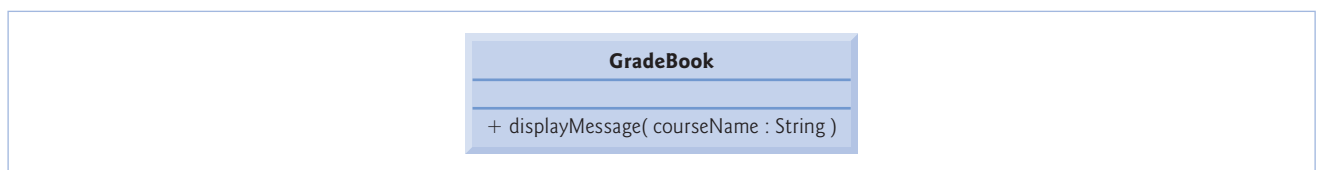


Figura 3.4 Diagrama de classes UML indicando que a classe `GradeBook` tem uma operação `displayMessage` com um parâmetro `courseName` do tipo UML `String`.

de dados e são declaradas dentro de uma definição de classe, mas fora dos corpos das definições de função-membro da classe. Todo objeto de uma classe mantém sua própria cópia de seus atributos na memória. O exemplo nesta seção demonstra uma classe `GradeBook` que contém um membro de dados `courseName` para representar o nome do curso de um objeto `GradeBook` particular.

A classe **`GradeBook`** com um membro de dados, uma função set e uma função get

No nosso próximo exemplo, a classe `GradeBook` (Figura 3.5) mantém o nome do curso como um membro de dados para que ele possa ser utilizado ou modificado a qualquer hora durante a execução de um programa. A classe contém as funções-membro `setCourseName`, `getCourseName` e `displayMessage`. A função-membro `setCourseName` armazena um nome do curso em um membro de dados `GradeBook` — a função-membro `getCourseName` obtém desse membro de dados o nome do curso de um `GradeBook`. A função-membro `displayMessage` — que agora não especifica nenhum parâmetro — ainda exibe uma mensagem de boas-vindas que inclui o nome do curso. Entretanto, como você verá, a função-membro agora obtém o nome do curso chamando outra função na mesma classe — `getCourseName`.

```

1 // Figura 3.5: fig03_05.cpp
2 // Define a classe GradeBook que contém um membro de dados courseName
3 // e funções-membro para configurar e obter seu valor;
4 // Cria e manipula um objeto GradeBook com essas funções.
5 #include <iostream>
6 using std::cout;
7 using std::cin;
8 using std::endl;
9
10 #include <string> // o programa utiliza classe de string padrão C++
11 using std::string;
12 using std::getline;
13
14 // definição da classe GradeBook
15 class GradeBook
16 {
17 public:
18     // função que configura o nome do curso
19     void setCourseName( string name )
20     {
21         courseName = name; // armazena o nome do curso no objeto
22     } // fim da função setCourseName
23
24     // função que obtém o nome do curso
25     string getCourseName()
26     {
27         return courseName; // retorna o courseName do objeto
28     } // fim da função getCourseName
29
30     // função que exibe uma mensagem de boas-vindas
31     void displayMessage()
32     {
33         // essa instrução chama getCourseName para obter o
34         // nome do curso que esse GradeBook representa
35         cout << "Welcome to the grade book for\n" << getCourseName() << "!"
36             << endl;
37     } // fim da função displayMessage
38 private:
39     string courseName; // nome do curso para esse GradeBook
40 }; // fim da classe GradeBook
41

```

Figura 3.5 Definindo e testando a classe `GradeBook` com um membro de dados e as funções set e get.

(continua)

```

42 // a função main inicia a execução do programa
43 int main()
44 {
45     string nameOfCourse; // strings de caracteres para armazenar o nome do curso
46     GradeBook myGradeBook; // cria um objeto GradeBook chamado myGradeBook
47
48     // exibe valor inicial de courseName
49     cout << "Initial course name is: " << myGradeBook.getCourseName()
50         << endl;
51
52     // solicita, insere e configura o nome do curso
53     cout << "\nPlease enter the course name:" << endl;
54     getline( cin, nameOfCourse ); // lê o nome de um curso com espaços em branco
55     myGradeBook.setCourseName( nameOfCourse ); // configura o nome do curso
56
57     cout << endl; // gera saída de uma linha em branco
58     myGradeBook.displayMessage(); // exibe a mensagem com o novo nome do curso
59     return 0; // indica terminação bem-sucedida
60 } // fim de main

```

Initial course name is:

Please enter the course name:

CS101 Introduction to C++ Programming

Welcome to the grade book for

CS101 Introduction to C++ Programming!

Figura 3.5 Definindo e testando a classe GradeBook com um membro de dados e as funções set e get.

(continuação)



Boa prática de programação 3.3

Coloque uma linha em branco entre definições de função-membro para aprimorar a legibilidade do programa.

Um instrutor típico dá mais de um curso, cada um com seu próprio nome. A linha 39 declara que `courseName` é uma variável de tipo `string`. Como a variável é declarada na definição de classe (linhas 15–40), mas fora dos corpos das definições de função-membro da classe (linhas 19–22, 25–28 e 31–37), a linha 39 é uma declaração para um membro de dados. Cada instância (isto é, objeto) da classe `GradeBook` contém uma cópia de cada um dos membros de dados da classe. Por exemplo, se houver dois objetos `GradeBook`, cada objeto tem sua própria cópia de `courseName` (uma por objeto), como veremos no exemplo da Figura 3.7. Um benefício de tornar `courseName` um membro de dados é que todas as funções-membro da classe (nesse caso, `GradeBook`) podem manipular quaisquer membros de dados que aparecerem na definição de classe (nesse caso, `courseName`).

Especificadores de acesso *public* e *private*

A maioria das declarações de membro de dados aparece depois do rótulo de especificador de acesso **private**: (linha 38). Como **public**, a palavra-chave **private** é um especificador de acesso. Variáveis ou funções declaradas depois do especificador de acesso **private** (e antes do próximo especificador de acesso) são acessíveis somente a funções-membro da classe para a qual elas são declaradas. Portanto, o membro de dados `courseName` pode ser utilizado somente em funções-membro `setCourseName`, `getCourseName` e `displayMessage` (de cada objeto) da classe `GradeBook`. O membro de dados `courseName`, porque é **private**, não pode ser acessado por funções fora da classe (como `main`) ou por funções-membro de outras classes no programa. Tentar acessar o membro de dados `courseName` em uma dessas localizações de programa com uma expressão como `myGradeBook.courseName` resultaria em um erro de compilação contendo uma mensagem semelhante a

```
cannot access private member declared in class 'GradeBook'
```



Observação de engenharia de software 3.1

*Como regra geral, os membros de dados devem ser declarados **private** e as funções-membro devem ser declaradas **public**. (Veremos que é apropriado declarar certas funções-membro **private**, se elas precisarem ser acessadas somente por outras funções-membro da classe.)*



Erro comum de programação 3.6

Uma tentativa de uma função, que não seja um membro de uma classe particular (ou um friend dessa classe, como veremos no Capítulo 10), de acessar um membro `private` dessa classe é um erro de compilação.

O acesso-padrão de membros de classe é `private`, então todos os membros depois do cabeçalho de classe e antes do primeiro especificador de acesso são `private`. Os especificadores de acesso `public` e `private` podem ser repetidos, mas isso é desnecessário e pode ser confuso.



Boa prática de programação 3.4

Apesar de os especificadores de acesso `public` e `private` poderem ser repetidos e combinados, liste todos os membros `public` de uma classe primeiro em um grupo e, então, liste todos os membros `private` em outro grupo. Isso focaliza a atenção do cliente na interface `public` da classe, em vez de na implementação da classe.



Boa prática de programação 3.5

Se escolher listar os membros `private` primeiro em uma definição de classe, utilize o especificador de acesso `private` explicitamente apesar de `private` ser assumido por padrão. Isso melhora a clareza do programa.

Declarar membros de dados com o especificador de acesso `private` é conhecido como **ocultação de dados**. Quando um programa cria (instancia) um objeto da classe `GradeBook`, o membro de dados `courseName` é encapsulado (ocultado) no objeto e pode ser acessado apenas por funções-membro da classe do objeto. Na classe `GradeBook`, as funções-membro `setCourseName` e `getCourseName` manipulam o membro de dados `courseName` diretamente (e `displayMessage` poderia fazer isso se necessário).



Observação de engenharia de software 3.2

Aprenderemos no Capítulo 10, “Classes: Um exame mais profundo, parte 2”, que funções e classes declaradas por uma classe como `friends` podem acessar os membros `private` da classe.



Dica de prevenção de erro 3.1

Transformar os membros de dados de uma classe `private` e as funções-membro da classe `public` facilita a depuração porque os problemas com manipulações de dados são localizados para as funções-membro da classe ou para os `friends` da classe.

As funções-membro `setCourseName` e `getCourseName`

A função-membro `setCourseName` (definida nas linhas 19–22) não retorna nenhum dado quando completa sua tarefa, então seu tipo de retorno é `void`. A função-membro recebe um parâmetro — `name` — que representa o nome do curso que será passado para ele como um argumento (como veremos na linha 55 de `main`). A linha 21 atribui `name` ao membro de dados `courseName`. Nesse exemplo, `setCourseName` não tenta validar o nome do curso — isto é, a função não verifica se o nome do curso obedece a algum formato particular ou segue alguma outra regra relacionada com o que seria um nome do curso ‘válido’. Suponha, por exemplo, que uma universidade possa imprimir listas de alunos contendo nomes de curso de apenas 25 caracteres ou menos. Nesse caso, talvez queiramos que a classe `GradeBook` assegure que seu membro de dados `courseName` nunca contenha mais de 25 caracteres. Discutimos as técnicas básicas de validação na Seção 3.10.

A função-membro `getCourseName` (definida nas linhas 25–28) retorna um `courseName` de um objeto `GradeBook` particular. A função-membro tem uma lista de parâmetros vazia, portanto não requer dados adicionais para realizar sua tarefa. A função especifica que ela retorna uma `string`. Quando uma função que especifica um tipo de retorno diferente de `void` é chamada e completa sua tarefa, a função retorna um resultado para sua função chamadora. Por exemplo, ao utilizar um caixa eletrônico (*automated teller machine* – ATM) e solicitar o saldo da sua conta, você espera o ATM devolver um valor que representa seu saldo. De modo semelhante, quando uma instrução chama a função-membro `getCourseName` em um objeto `GradeBook`, a instrução espera receber o nome do curso do `GradeBook` (nesse caso, uma `string`, como especificado pelo tipo de retorno da função). Se tiver uma função `square` que retorna o quadrado de seu argumento, a instrução

```
int result = square( 2 );
```

retorna 4 a partir da função `square` e inicializa a variável `result` com o valor 4. Se você tiver uma função `maximum` que retorna o maior de três argumentos de inteiro, a instrução

```
int biggest = maximum( 27, 114, 51 );
```

retorna 114 a partir da função `maximum` e inicializa a variável `biggest` com o valor 114.



Erro comum de programação 3.7

Esquecer de retornar um valor de uma função que supostamente deve retornar um valor é um erro de compilação.

Observe que ambas as instruções nas linhas 21 e 27 utilizam a variável `courseName` (linha 39) embora ela não tenha sido declarada em nenhuma das funções-membro. Podemos utilizar `courseName` nas funções-membro da classe `GradeBook` porque `courseName` é um membro de dados da classe. Observe também que a ordem em que funções-membro são definidas não determina quando elas são chamadas em tempo de execução. Então a função-membro `getCourseName` poderia ser definida antes da função-membro `setCourseName`.

Função-membro `displayMessage`

A função-membro `displayMessage` (linhas 31–37) não retorna dados quando ela completa sua tarefa, então seu tipo de retorno é `void`. A função não recebe parâmetros, então sua lista de parâmetros está vazia. As linhas 35–36 geram saída de uma mensagem de boas-vindas que inclui o valor do membro de dados `courseName`. A linha 35 chama a função-membro `getCourseName` para obter o valor de `courseName`. Observe que a função-membro `displayMessage` também poderia acessar diretamente o membro de dados `courseName`, exatamente como as funções-membro `setCourseName` e `getCourseName` acessam. Explicaremos em breve a razão pela qual escolhemos chamar a função-membro `getCourseName` para obter o valor de `courseName`.

Testando a classe `GradeBook`

A função `main` (linhas 43–60) cria um objeto da classe `GradeBook` e utiliza cada uma de suas funções-membro. A linha 46 cria um objeto `GradeBook` chamado `myGradeBook`. As linhas 49–50 exibem o nome inicial de curso chamando a função-membro `getCourseName` do objeto. Observe que a primeira linha da saída não mostra um nome do curso, porque o membro de dados `courseName` do objeto (isto é, uma `string`) está inicialmente vazio — por padrão, o valor inicial de uma `string` é a chamada **string vazia**, isto é, uma `string` que não contém nenhum caractere. Nada aparece na tela quando uma `string` vazia é exibida.

A linha 53 exibe um prompt que pede para o usuário inserir o nome de um curso. A variável `string` local `nameOfCourse` (declarada na linha 45) é configurada como o nome do curso inserido pelo usuário, que é obtido pela chamada à função `getline` (linha 54). A linha 55 chama a função-membro `setCourseName` do objeto `myGradeBook` e fornece `nameOfCourse` como o argumento da função. Quando a função é chamada, o valor do argumento é copiado no parâmetro `name` (linha 19) da função-membro `setCourseName` (linhas 19–22). Então o valor do parâmetro é atribuído ao membro de dados `courseName` (linha 21). A linha 57 pula uma linha na saída; então a linha 58 chama a função-membro `displayMessage` do objeto `myGradeBook` para exibir a mensagem de boas-vindas contendo o nome do curso.

Engenharia de software com as funções `set` e `get`

Os membros de dados `private` de uma classe só podem ser manipulados por funções-membro dessa classe (e por ‘amigos’ da classe, como veremos no Capítulo 10, “Classes: um exame mais profundo, parte 2”). Portanto, um cliente de um objeto — isto é, qualquer classe ou função que chame as funções-membro do objeto de fora do objeto — chama as funções-membro `public` da classe para solicitar os serviços dessa classe para objetos particulares dela própria. Essa é a razão pela qual as instruções na função `main` (Figura 3.5, linhas 43–60) chamam as funções-membro `setCourseName`, `getCourseName` e `displayMessage` em um objeto `GradeBook`. As classes costumam fornecer funções-membro `public` para permitir a clientes da classe *configurar* (*set*, isto é, atribuir valores a) ou *obter* (*get*, isto é, obter valores de) membros de dados `private`. Os nomes dessas funções-membro não precisam iniciar com *set* ou *get*, mas essa convenção de atribuição de nomes é comum. Nesse exemplo, a função-membro que *configura* (*set*) o membro de dados `courseName` é chamada `setCourseName` e a função-membro que *obtem* (*get*) o valor do membro de dados `courseName` é chamada `getCourseName`. Observe que as funções *set* também são às vezes chamadas de **modificadoras** (porque modificam valores), e as funções *get* também são às vezes chamadas de **funções de acesso** (porque acessam valores).

Lembre-se de que declarar membros de dados com o especificador de acesso `private` impõe a ocultação de dados. Fornecer funções *set* e *get* `public` permite aos clientes de uma classe acessar os dados ocultos, mas apenas *indiretamente*. O cliente sabe que está tentando modificar ou obter os dados de um objeto, mas o cliente não sabe como o objeto realiza essas operações. Em alguns casos, uma classe pode representar internamente uma parte de dados de uma maneira, mas expõe esses dados para clientes de maneira diferente. Por exemplo, suponha que uma classe `Clock` represente a hora do dia como um membro de dados `private int time` que armazena o número de segundos desde a meia-noite. Entretanto, quando um cliente chama a função-membro `getTime` de um objeto `Clock`, o objeto poderia retornar o tempo com horas, minutos e segundos em uma `string` no formato “HH:MM:SS”. De modo semelhante, suponha que a classe `Clock` forneça uma função *set* chamada `setTime` que aceita um parâmetro `string` no formato “HH:MM:SS”. Utilizando as capacidades de `string` apresentadas no Capítulo 18, a função `setTime` poderia converter essa `string` em um número de segundos, que a função armazena em seu membro de dados `private`. A função *set* também poderia verificar se o valor que ela recebe representa uma hora válida (por exemplo, “12:30:45” é válida, mas “42:85:70” não é). As funções *set* e *get* permitem a um cliente interagir com um objeto, mas os dados `private` do objeto permanecem seguramente encapsulados (isto é, ocultos) no próprio objeto.

As funções *set* e *get* de uma classe também devem ser utilizadas por outras funções-membro dentro da classe para manipular os dados `private` da classe, embora essas funções-membro *possam* acessar os dados `private` diretamente. Na Figura 3.5, as funções-membro `setCourseName` e `getCourseName` são funções-membro `public`, portanto acessíveis aos clientes da classe, bem como à própria classe. A função-membro `displayMessage` chama a função-membro `getCourseName` para obter o valor do membro de dados `courseName` para propósitos de exibição, mesmo que `displayMessage` possa acessar `courseName` diretamente — acessar um membro de dados via sua função *get* cria uma classe melhor e mais robusta (isto é, uma classe mais fácil de manter e com menos probabilidade de parar de

funcionar). Se decidirmos alterar o membro de dados `courseName` de alguma maneira, a definição `displayMessage` não exigirá modificação — apenas os corpos das funções `get` e `set` que manipulam diretamente o membro de dados precisam mudar. Por exemplo, suponha que decidíssemos representar o nome do curso como dois membros de dados separados — `courseNumber` (por exemplo, "CS101") e `courseTitle` (por exemplo, "Introduction to C++ Programming"). A função-membro `displayMessage` ainda pode emitir uma única chamada para a função-membro `getCourseName` a fim de obter o nome inteiro do curso exibido como parte da mensagem de boas-vindas. Nesse caso, `getCourseName` precisaria construir e retornar uma `string` contendo o `courseNumber` seguido pelo `courseTitle`. A função-membro `displayMessage` continuaria a exibir o título completo do curso 'CS101 Introduction to C++ Programming', porque ela não é afetada pela alteração nos membros de dados da classe. Os benefícios de chamar uma função `set` de outra função-membro de uma classe se tornarão claros quando discutirmos validação na Seção 3.10.



Boa prática de programação 3.6

Tente sempre localizar os efeitos de alterações em membros de dados de uma classe acessando e manipulando os membros de dados por meio de suas funções `get` e `set`. Alterações no nome de um membro de dados ou tipo de dados utilizado para armazenar um membro de dados afetam então apenas as funções `get` e `set` correspondentes, mas não os chamadores dessas funções.



Observação de engenharia de software 3.3

É importante escrever programas que sejam compreensíveis e fáceis de manter. A mudança é a regra em vez da exceção. Os programadores devem antecipar que seu código será modificado.



Observação de engenharia de software 3.4

O designer de classes não precisa fornecer as funções `get` e `set` para cada item de dados `private`; essas capacidades devem ser fornecidas somente quando apropriado. Se um serviço é útil para o código-cliente, esse serviço deve, em geral, ser fornecido na interface `public` da classe.

O diagrama de classes UML **GradeBook** com um membro de dados e as funções `get` e `set`

A Figura 3.6 contém um diagrama de classes UML atualizada da versão da classe `GradeBook` na Figura 3.5. Esse diagrama modela o membro de dados `courseName` da classe `GradeBook` como um atributo no compartimento do meio da classe. A UML representa os membros de dados como atributos listando o nome do atributo, seguido por um caractere de dois-pontos e pelo tipo do atributo. O tipo UML do atributo `courseName` é `String`, que corresponde à `string` em C++. O membro de dados `courseName` é `private` em C++, então o diagrama de classes lista um sinal de subtração (–) na frente do nome do atributo correspondente. O sinal de subtração na UML é equivalente ao especificador de acesso `private` em C++. A classe `GradeBook` contém três funções-membro `public`, então o diagrama de classes lista três operações no terceiro compartimento. Lembre-se de que o sinal de adição (+) antes de cada nome de operação indica que a operação é `public` no C++. A operação `setCourseName` tem um parâmetro `String` chamado `name`. A UML indica o tipo de retorno de uma operação colocando dois-pontos e o tipo de retorno depois dos parênteses que se seguem ao nome da operação. A função-membro `getCourseName` da classe `GradeBook` (Figura 3.5) tem um tipo de retorno `string` em C++, portanto o diagrama de classes mostra um tipo de retorno `String` na UML. Observe que as operações `setCourseName` e `displayMessage` não retornam valores (isto é, elas retornam `void`), então o diagrama de classes UML não especifica um tipo de retorno depois dos parênteses dessas operações. A UML não utiliza `void` como o C++ utiliza quando uma função não retorna um valor.

3.7 Inicializando objetos com construtores

Como mencionado na Seção 3.6, quando um objeto da classe `GradeBook` (Figura 3.5) é criado, seu membro de dados `courseName` é inicializado, por padrão, como `string` vazia. E se você quiser fornecer o nome de um curso quando criar um objeto `GradeBook`? Cada classe que você declara pode fornecer um **construtor** que pode ser utilizado para inicializar um objeto de uma classe quando o objeto é criado. Um construtor é uma função-membro especial que deve ser definida com o mesmo nome da classe, de modo que o compi-

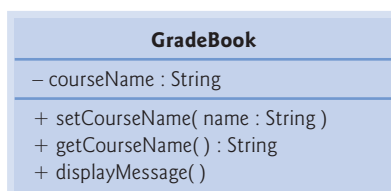


Figura 3.6 Diagrama de classes UML para a classe `GradeBook` com um atributo privado `courseName` e operações públicas `setCourseName`, `getCourseName` e `displayMessage`.

lador possa diferenciá-lo de outras funções-membro da classe. Uma diferença importante entre construtores e outras funções é que os construtores não podem retornar valores, então eles não podem especificar um tipo de retorno (nem mesmo `void`). Normalmente, os construtores são declarados `public`. O termo ‘construtor’ é, freqüentemente, abreviado como ‘ctor’ na literatura — mas preferimos não utilizar essa abreviação.

O C++ requer uma chamada de construtor para cada objeto que é criado, o que ajuda a assegurar que o objeto é inicializado adequadamente antes de ser utilizado em um programa — a chamada de construtor ocorre implicitamente quando o objeto é criado. Em qualquer classe que não inclua um construtor explicitamente, o compilador fornece um **construtor-padrão** — isto é, um construtor sem parâmetros. Por exemplo, quando a linha 46 da Figura 3.5 cria um objeto `GradeBook`, o construtor-padrão é chamado, porque a declaração de `myGradeBook` não especifica nenhum argumento de construtor. O construtor-padrão fornecido pelo compilador cria um objeto `GradeBook` sem fornecer nenhum valor inicial para os membros de dados do objeto. [Nota: Para os membros de dados que são objetos de outras classes, o construtor-padrão chama o construtor-padrão de cada membro de dados para assegurar que o membro de dados seja inicializado de maneira adequada. De fato, essa é a razão pela qual o membro de dados `string courseName` (na Figura 3.5) foi inicializado como uma string vazia — o construtor-padrão da classe `string` configura o valor da string como a string vazia. Na Seção 10.3 você aprenderá mais sobre como inicializar membros de dados que são objetos de outras classes.]

No exemplo da Figura 3.7 especificamos um nome do curso para um objeto `GradeBook` quando o objeto é criado (linha 49). Nesse caso, o argumento “CS101 Introduction to C++ Programming” é passado para o construtor do objeto `GradeBook` (linhas 17–20) e utilizado para inicializar `courseName`. A Figura 3.7 define uma classe `GradeBook` modificada contendo um construtor com um parâmetro `string` que recebe o nome do curso inicial.

```

1  // Figura 3.7: fig03_07.cpp
2  // Instanciando múltiplos objetos da classe GradeBook e utilizando
3  // o construtor GradeBook para especificar o nome do curso
4  // quando cada objeto GradeBook é criado.
5  #include <iostream>
6  using std::cout;
7  using std::endl;
8
9  #include <string> // o programa utiliza classe de string padrão C++
10 using std::string;
11
12 // Definição da classe GradeBook
13 class GradeBook
14 {
15 public:
16     // o construtor inicializa courseName com a string fornecida como argumento
17     GradeBook( string name )
18     {
19         setCourseName( name ); // chama a função set para inicializar courseName
20     } // fim do construtor GradeBook
21
22     // função para configurar o nome do curso
23     void setCourseName( string name )
24     {
25         courseName = name; // armazena o nome do curso no objeto
26     } // fim da função setCourseName
27
28     // função para obter o nome do curso
29     string getCourseName()
30     {
31         return courseName; // retorna courseName do objeto
32     } // fim da função getCourseName
33
34     // exibe uma mensagem de boas-vindas para o usuário GradeBook
35     void displayMessage()

```

Figura 3.7 Instanciando múltiplos objetos da classe `GradeBook` e utilizando o construtor `GradeBook` para especificar o nome do curso quando cada objeto `GradeBook` é criado. (continua)

```

36     {
37         // chama getCourseName para obter o courseName
38         cout << "Welcome to the grade book for\n" << getCourseName()
39             << "!" << endl;
40     } // fim da função displayMessage
41 private:
42     string courseName; // nome do curso para esse GradeBook
43 }; // fim da classe GradeBook
44
45 // a função main inicia a execução do programa
46 int main()
47 {
48     // cria dois objetos GradeBook
49     GradeBook gradeBook1( "CS101 Introduction to C++ Programming" );
50     GradeBook gradeBook2( "CS102 Data Structures in C++" );
51
52     // exibe valor inicial de courseName para cada GradeBook
53     cout << "gradeBook1 created for course: " << gradeBook1.getCourseName()
54         << "\ngradeBook2 created for course: " << gradeBook2.getCourseName()
55         << endl;
56     return 0; // indica terminação bem-sucedida
57 } // fim de main

```

```

gradeBook1 created for course: CS101 Introduction to C++ Programming
gradeBook2 created for course: CS102 Data Structures in C++

```

Figura 3.7 Instanciando múltiplos objetos da classe GradeBook e utilizando o construtor GradeBook para especificar o nome do curso quando cada objeto GradeBook é criado. (continuação)

Definindo um construtor

As linhas 17–20 da Figura 3.7 definem um construtor para a classe GradeBook. Note que o construtor tem o mesmo nome que sua classe, GradeBook. Um construtor especifica em sua lista de parâmetros os dados que ele requer para realizar sua tarefa. Quando você cria um novo objeto, coloca esses dados entre os parênteses que se seguem ao nome de objeto (como fizemos nas linhas 49–50). A linha 17 indica que o construtor da classe GradeBook tem um parâmetro string chamado name. Observe que a linha 17 não especifica um tipo de retorno, porque os construtores não podem retornar valores (ou até mesmo void).

A linha 19 no corpo do construtor passa o parâmetro do construtor name para a função-membro setCourseName, que atribui um valor para membro de dados courseName. A função-membro setCourseName (linhas 23–26) simplesmente atribui seu parâmetro name ao membro de dados courseName, então talvez você se pergunte por que nos incomodamos em fazer a chamada para setCourseName na linha 19 — o construtor certamente poderia realizar a atribuição courseName = name. Na Seção 3.10, modificamos setCourseName para realizar a validação (assegurando que, nesse caso, o courseName tenha 25 ou menos caracteres de comprimento). Nesse ponto, os benefícios de chamar setCourseName a partir do construtor se tornarão claros. Observe que tanto o construtor (linha 17) como a função setCourseName (linha 23) utilizam um parâmetro chamado name. Você pode utilizar os mesmos nomes de parâmetro em funções diferentes porque os parâmetros são locais a cada função; um não interfere no outro.

Testando a classe GradeBook

As linhas 46–57 da Figura 3.7 definem a função main que testa a classe GradeBook e demonstra a inicialização de objetos GradeBook utilizando um construtor. A linha 49 na função main cria e inicializa um objeto GradeBook chamado gradeBook1. Quando essa linha executa, o construtor GradeBook (linhas 17–20) é chamado (implicitamente por C++) com o argumento "CS101 Introduction to C++ Programming" para inicializar o nome do curso do gradeBook1. A linha 50 repete esse processo para o objeto GradeBook chamado gradeBook2, dessa vez passando o argumento "CS102 Data Structures in C++" para inicializar o nome do curso do gradeBook2. As linhas 53–54 utilizam a função-membro getCourseName de cada objeto para obter os nomes de curso e mostra que eles, de fato, foram inicializados quando os objetos foram criados. A saída confirma que cada objeto GradeBook mantém sua própria cópia de membro de dados courseName.

Duas maneiras de fornecer um construtor-padrão para uma classe

Qualquer construtor que não aceita argumentos é chamado construtor-padrão. Uma classe obtém um construtor-padrão em uma de duas maneiras:

1. O compilador cria implicitamente um construtor-padrão em uma classe que não define um construtor. Esse construtor-padrão não inicializa os membros de dados da classe, mas chama o construtor-padrão para cada membro de dados que é um objeto de outra classe. [Nota: Uma variável não inicializada contém, em geral, um valor ‘lixo’ (por exemplo, uma variável `int` não inicializada que poderia conter -858993460, que provavelmente é um valor incorreto para essa variável na maioria dos programas).]
2. O programador define explicitamente um construtor que não aceita argumentos. Tal construtor-padrão realizará a inicialização especificada pelo programador e chamará o construtor-padrão para cada membro de dados que for um objeto de outra classe.

Se o programador definir um construtor com argumentos, C++ não criará implicitamente um construtor-padrão para essa classe. Observe que para cada versão da classe `GradeBook` na Figura 3.1, Figura 3.3 e Figura 3.5 o compilador definiu implicitamente um construtor-padrão.



Dica de prevenção de erro 3.2

A menos que nenhuma inicialização de membros de dados da classe seja necessária (quase nunca), forneça um construtor para assegurar que os membros de dados da classe sejam inicializados com valores significativos quando cada novo objeto da classe for criado.



Observação de engenharia de software 3.5

Os membros de dados podem ser inicializados em um construtor da classe ou seus valores podem ser configurados mais tarde depois que o objeto for criado. Entretanto, é uma boa prática de engenharia de software assegurar que um objeto seja completamente inicializado antes de o código-cliente invocar as funções-membro do objeto. Em geral, você não deve contar com o código-cliente para assegurar que um objeto seja inicializado adequadamente.

Adicionando o construtor ao diagrama de classes UML da classe `GradeBook`

O diagrama de classes UML da Figura 3.8 modela a classe `GradeBook` da Figura 3.7, que tem um construtor com um parâmetro `name` de tipo `string` (representado pelo tipo `String` na UML). Assim como operações, a UML modela construtores no terceiro compartimento de uma classe em um diagrama de classes. Para distinguir entre um construtor e operações de uma classe, a UML coloca a palavra ‘constructor’ entre aspas francesas (« e ») antes do nome do construtor. É comum listar o construtor da classe antes de outras operações no terceiro compartimento.

3.8 Colocando uma classe em um arquivo separado para reusabilidade

Desenvolvemos a classe `GradeBook` até o ponto em que precisávamos, por enquanto de uma perspectiva de programação, então vamos considerar algumas questões de engenharia de software. Um dos benefícios de criar definições de classes é que, quando empacotadas adequadamente, nossas classes podem ser reutilizadas por programadores — potencialmente em todo o mundo. Por exemplo, podemos reutilizar o tipo `string` da C++ Standard Library em qualquer programa C++ por meio da inclusão do arquivo de cabeçalho `<string>` no programa (e, como veremos, por meio da capacidade de vincular ao código-objeto da biblioteca).

Infelizmente, os programadores que desejam utilizar nossa classe `GradeBook` não podem simplesmente incluir o arquivo da Figura 3.7 em outro programa. Como você aprendeu no Capítulo 2, a função `main` começa a execução de cada programa e cada programa deve ter exatamente uma função `main`. Se outros programadores incluírem o código da Figura 3.7, obterão uma bagagem extra — nossa função `main` — e os programas terão, portanto, duas funções `main`. Quando eles tentarem compilar os programas, o compilador indicará um erro porque, novamente, cada programa pode ter apenas uma função `main`. Por exemplo, tentar compilar um programa com duas funções `main` no Microsoft Visual C++ .NET produz o erro

```
error C2084: function 'int main(void)' already has a body
```

quando o compilador tentar compilar a segunda função `main` que ele encontra. De modo semelhante, o compilador GNU C++ produz o erro

```
redefinition of 'int main()'
```

Esses erros indicam que um programa já tem uma função `main`. Portanto, colocar `main` no mesmo arquivo com uma definição de classe evita que a classe seja reutilizada por outros programas. Nesta seção, demonstramos como tornar a classe `GradeBook` reutilizável separando-a da função `main` dentro de outro arquivo.

Arquivos de cabeçalho

Cada um dos exemplos anteriores no capítulo consiste em um único arquivo `.cpp`, também conhecido como **arquivo de código-fonte**, que contém uma definição de classe `GradeBook` e uma função `main`. Ao construir um programa C++ orientado a objetos, é comum definir o código-fonte reutilizável (como uma classe) em um arquivo que, por convenção, tem uma extensão de nome de arquivo `.h` — conhecido como **arquivo de cabeçalho**. Os programas utilizam as diretivas de pré-processador `#include` para incluir arquivos de

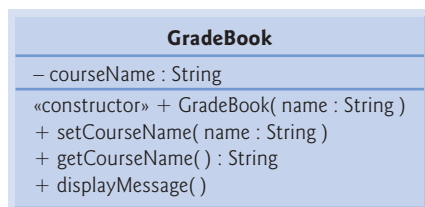


Figura 3.8 Diagrama de classes UML indicando que a classe `GradeBook` tem um construtor com um parâmetro `name` de tipo UML `String`.

cabeçalho e tirar proveito de componentes de softwares reutilizáveis, como o tipo `string` fornecido na C++ Standard Library e os tipos definidos pelo usuário, como a classe `GradeBook`.

Em nosso próximo exemplo, separamos o código da Figura 3.7 em dois arquivos — `GradeBook.h` (Figura 3.9) e `fig03_10.cpp` (Figura 3.10). Ao examinar o arquivo de cabeçalho na Figura 3.9, note que ele contém apenas a definição de classe `GradeBook` (linhas 11–41) e as linhas 3–8, o que permite que a classe `GradeBook` utilize o tipo `cout`, `endl` e `string`. A função `main` que utiliza a classe `GradeBook` é definida no arquivo de código-fonte `fig03_10.cpp` (Figura 3.10) nas linhas 10–21. Para ajudá-lo a se preparar para os programas maiores que você encontrará mais adiante neste livro e na indústria, costumamos utilizar um arquivo de código-fonte separado contendo a função `main` para testar nossas classes (isso é chamado de **programa driver**). Você logo aprenderá como um arquivo de código-fonte com `main` pode utilizar a definição de classe localizada em um arquivo de cabeçalho para criar objetos de uma classe.

```

1 // Figura 3.9: GradeBook.h
2 // Definição de classe GradeBook em um arquivo main separado.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string> // a classe GradeBook utiliza a classe de string padrão C++
8 using std::string;
9
10 // definição da classe GradeBook
11 class GradeBook
12 {
13 public:
14     // o construtor inicializa courseName com a string fornecida como argumento
15     GradeBook( string name )
16     {
17         setCourseName( name ); // chama a função set para inicializar courseName
18     } // fim do construtor GradeBook
19
20     // função para configurar o nome do curso
21     void setCourseName( string name )
22     {
23         courseName = name; // armazena o nome do curso no objeto
24     } // fim da função setCourseName
25
26     // função para obter o nome do curso
27     string getCourseName()
28     {
29         return courseName; // retorna courseName do objeto
30     } // fim da função getCourseName
31
32     // exibe uma mensagem de boas-vindas para o usuário GradeBook
33     void displayMessage()
  
```

Figura 3.9 A definição da classe `GradeBook`.

(continua)


```

34     {
35         // chama getCourseName para obter o courseName
36         cout << "Welcome to the grade book for\n" << getCourseName()
37             << "!" << endl;
38     } // fim da função displayMessage
39 private:
40     string courseName; // nome do curso para esse GradeBook
41 }; // fim da classe GradeBook

```

Figura 3.9 A definição da classe GradeBook.

(continuação)

Incluindo um arquivo de cabeçalho que contém uma classe definida pelo usuário

Um arquivo de cabeçalho como GradeBook.h (Figura 3.9) não pode ser utilizado para iniciar a execução de programa, porque ele não contém uma função main. Se você tentar compilar e vincular GradeBook.h por conta própria para criar um aplicativo executável, o Microsoft Visual C++ .NET produzirá a mensagem de erro de linker:

```
error LNK2019: unresolved external symbol _main referenced in function _mainCRTStartup
```

Executar GNU C++ no Linux produz uma mensagem de erro de linker contendo:

```
undefined reference to 'main'
```

Esse erro indica que o linker não pôde localizar a função main do programa. Para testar a classe GradeBook (definida na Figura 3.9), você deve escrever um arquivo de código-fonte separado contendo uma função main (como Figura 3.10) que instancia e utiliza os objetos da classe.

A partir da discussão da Seção 3.4, lembre-se de que, enquanto o compilador sabe o que são tipos de dados fundamentais como int, o compilador não sabe o que é um GradeBook porque ele é um tipo definido pelo usuário. De fato, o compilador nem mesmo conhece as classes na C++ Standard Library. Para ajudar a entender como utilizar uma classe, devemos fornecer explicitamente o compilador com a definição da classe — essa é a razão pela qual, por exemplo, para utilizar o tipo string, um programa deve incluir o arquivo de cabeçalho <string>. Isso permite ao compilador determinar a quantidade de memória que deve reservar para cada objeto da classe e assegura que um programa chame corretamente as funções-membro da classe.

```

1 // Figura 3.10: fig03_10.cpp
2 // Incluindo a classe GradeBook a partir do arquivo Gradebook.h para uso em main.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "GradeBook.h" // inclui a definição de classe GradeBook
8
9 // a função main inicia a execução do programa
10 int main()
11 {
12     // cria dois objetos GradeBook
13     GradeBook gradeBook1( "CS101 Introduction to C++ Programming" );
14     GradeBook gradeBook2( "CS102 Data Structures in C++" );
15
16     // exibe valor inicial de courseName para cada GradeBook
17     cout << "gradeBook1 created for course: " << gradeBook1.getCourseName()
18         << "\ngradeBook2 created for course: " << gradeBook2.getCourseName()
19         << endl;
20     return 0; // indica terminação bem-sucedida
21 } // fim de main

```

```

gradeBook1 created for course: CS101 Introduction to C++ Programming
gradeBook2 created for course: CS102 Data Structures in C++

```

Figura 3.10 Incluindo a classe GradeBook a partir do arquivo GradeBook.h para utilizar em main.

Para criar os objetos `GradeBook` `gradeBook1` e `gradeBook2` nas linhas 13–14 da Figura 3.10, o compilador deve saber o tamanho de um objeto `GradeBook`. Enquanto os objetos contêm conceitualmente membros de dados e funções-membro, os objetos C++, em geral, contêm apenas dados. O compilador cria apenas uma cópia das funções-membro da classe e compartilha essa cópia entre todos os objetos da classe. Cada objeto, naturalmente, precisa de sua própria cópia dos membros de dados da classe, porque seu conteúdo pode variar entre objetos (como dois objetos `BankAccount` diferentes tendo dois membros de dados `balance` diferentes). O código de função-membro, porém, não é modificável, podendo ser compartilhado entre todos os objetos da classe. Portanto, o tamanho de um objeto depende da quantidade de memória necessária para armazenar os membros de dados da classe. Incluindo `GradeBook.h` na linha 7, fornecemos para o compilador acesso às informações de que ele precisa (Figura 3.9, linha 40) para determinar o tamanho de um objeto `GradeBook` e determinar se os objetos da classe são utilizados corretamente (nas linhas 13–14 e 17–18 da Figura 3.10).

A linha 7 instrui o pré-processador C++ a substituir a diretiva por uma cópia do conteúdo de `GradeBook.h` (isto é, a definição de classe `GradeBook`) antes de o programa ser compilado. Quando o arquivo de código-fonte `fig03_10.cpp` está compilado, ele agora contém a definição de classe `GradeBook` (por causa do `#include`) e o compilador é capaz de determinar como criar os objetos `GradeBook` e ver que suas funções-membro são chamadas corretamente. Agora que a definição de classe está em um arquivo de cabeçalho (sem função `main`), você pode incluir esse cabeçalho em *qualquer* programa que precise reutilizar nossa classe `GradeBook`.

Como arquivos de cabeçalho são localizados

Note que o nome do arquivo de cabeçalho `GradeBook.h` na linha 7 da Figura 3.10 está entre aspas (" ") em vez de entre os colchetes angulares (< >). Normalmente, os arquivos de código-fonte e arquivos de cabeçalho definidos pelo usuário de um programa são colocados no mesmo diretório. Quando o pré-processador encontra um nome de arquivo de cabeçalho entre aspas (por exemplo, "`GradeBook.h`"), o pré-processador tenta localizar o arquivo de cabeçalho no mesmo diretório do arquivo em que a diretiva `#include` aparece. Se o pré-processador não puder localizar o arquivo de cabeçalho nesse diretório, ele o procura na(s) mesma(s) localização(ões) dos arquivos de cabeçalho C++ Standard Library. Quando o pré-processador encontra um nome de arquivo de cabeçalho entre colchetes angulares (por exemplo, `<iostream>`), ele assume que o cabeçalho faz parte da C++ Standard Library e não examina o diretório do programa que está sendo pré-processado.



Dica de prevenção de erro 3.3

Para assegurar que o pré-processador possa localizar os arquivos de cabeçalho corretamente, as diretivas de pré-processador `#include` devem colocar os nomes de arquivos de cabeçalho definidos pelo usuário entre aspas (por exemplo, "`GradeBook.h`") e colocar os nomes de arquivos de cabeçalho da C++ Standard Library entre colchetes angulares (por exemplo, `<iostream>`).

Questões de engenharia de software adicionais

Agora que a classe `GradeBook` foi definida em um arquivo de cabeçalho, a classe é reutilizável. Infelizmente, colocar uma definição de classe em um arquivo de cabeçalho como na Figura 3.9 ainda revela a inteira implementação da classe para os clientes da classe — `GradeBook.h` é simplesmente um arquivo de texto que qualquer pessoa pode abrir e ler. A sabedoria da engenharia de software convencional diz que, para utilizar um objeto de uma classe, o código-cliente precisa saber somente quais funções-membro chamar, quais argumentos fornecer para cada função-membro e qual tipo de retorno esperar de cada função-membro. O código-cliente não precisa saber como essas funções são implementadas.

Se o código-cliente sabe como uma classe é implementada, o programador de código-cliente poderia escrever o código-cliente com base nos detalhes de implementação da classe. Idealmente, se essa implementação mudar, os clientes da classe não devem ter de mudar. Ocultar os detalhes de implementação da classe facilita a alteração de implementação da classe, ao mesmo tempo que minimiza e, espera-se, elimina a alteração do código-cliente.

Na Seção 3.9, mostramos como dividir a classe `GradeBook` em dois arquivos de modo que

1. a classe seja reutilizável;
2. os clientes da classe saibam que funções-membro a classe fornece, como chamá-las e que tipos de retorno esperar;
3. os clientes não saibam como as funções-membro da classe foram implementadas.

3.9 Separando a interface da implementação

Na seção anterior mostramos como promover a reusabilidade de software separando uma definição de classe do código-cliente (por exemplo, função `main`) que utiliza a classe. Agora introduzimos outro princípio fundamental de boa engenharia de software — **separar a interface da implementação**.

Interface de uma classe

Interfaces definem e padronizam o modo como coisas, pessoas e sistemas interagem entre si. Por exemplo, os controles de um rádio servem como uma interface entre os usuários do rádio e seus componentes internos. Os controles permitem aos usuários realizar um conjunto limitado de operações (como alterar a estação, ajustar o volume e escolher entre estações AM e FM). Vários rádios podem implementar essas operações de diferentes modos — alguns fornecem botões, alguns fornecem sintonizadores e outros suportam co-

mandos de voz. A interface especifica que operações um rádio permite que os usuários realizem, mas não especifica como as operações estão implementadas dentro do rádio.

De maneira semelhante, a **interface de uma classe** descreve *que* serviços os clientes de uma classe podem utilizar e como *solicitar* esses serviços, mas não *como* a classe executa os serviços. A interface de uma classe consiste nas funções-membro `public` da classe (também conhecidas como **serviços public** da classe). Por exemplo, a interface da classe `GradeBook` (Figura 3.9) contém um construtor e as funções-membro `setCourseName`, `getCourseName` e `displayMessage`. Os clientes do `GradeBook` (por exemplo, `main` na Figura 3.10) utilizam essas funções para solicitar os serviços da classe. Como logo veremos, você pode especificar a interface de uma classe escrevendo uma definição de classe que lista apenas os nomes de função-membro, tipos de retorno e tipos de parâmetro.

Separando a interface da implementação

Em nossos primeiros exemplos, todas as definições de classe continham as definições completas das funções-membro `public` da classe e as declarações de seus membros de dados `private`. Entretanto, é uma melhor engenharia de software definir funções-membro fora da definição de classe, para que os detalhes da sua implementação possam ficar ocultos do código-cliente. Essa prática assegura que os programadores não escrevam código-cliente que dependa dos detalhes de implementação da classe. Se eles precisassem fazer isso, o código-cliente provavelmente ‘quebraria’ se a implementação da classe fosse alterada.

O programa das figuras 3.11–3.13 separa a interface da classe `GradeBook` de sua implementação dividindo a definição de classe da Figura 3.9 em dois arquivos — o arquivo de cabeçalho `GradeBook.h` (Figura 3.11) em que classe `GradeBook` é definida e o arquivo de código-fonte `GradeBook.cpp` (Figura 3.12) em que funções-membro de `GradeBook` são definidas. Por convenção, as definições de função-membro são colocadas em um arquivo de código-fonte com o mesmo nome de base (por exemplo, `GradeBook`) que o arquivo de cabeçalho da classe, mas com uma extensão do nome do arquivo `.cpp`. O arquivo de código-fonte `fig03_13.cpp` (Figura 3.13) define a função `main` (o código-cliente). O código e a saída da Figura 3.13 são idênticos àqueles da Figura 3.10. A Figura 3.14 mostra como esse programa de três arquivos é compilado a partir das perspectivas do programador de classe `GradeBook` e do programador do código-cliente — explicaremos essa figura em detalhes.

GradeBook.h: Definindo a interface de uma classe com protótipos de função

O arquivo de cabeçalho `GradeBook.h` (Figura 3.11) contém outra versão de definição de classe de `GradeBook` (linhas 9–18). Essa versão é semelhante à da Figura 3.9, mas as definições de função na Figura 3.9 são substituídas aqui por **protótipos de função** (linhas 12–15) que descrevem a interface `public` da classe sem revelar as implementações da função-membro da classe. Um protótipo de função é uma declaração de função que informa ao compilador o nome da função, seu tipo de retorno e os tipos de seus parâmetros. Observe que o arquivo de cabeçalho especifica também o membro de dados `private` da classe (linha 17). Novamente, o compilador deve conhecer os membros de dados da classe para determinar quanta memória reservar para cada objeto da classe. Incluir o arquivo de cabeçalho `GradeBook.h` no código-cliente (linha 8 da Figura 3.13) fornece ao compilador as informações de que ele precisa para assegurar que o código-cliente chame corretamente as funções-membro da classe `GradeBook`.

O protótipo de função na linha 12 (Figura 3.12) indica que o construtor requer um parâmetro `string`. Lembre-se de que os construtores não têm tipos de retorno, portanto nenhum tipo de retorno aparece no protótipo de função. O protótipo de função da função-membro `setCourseName` (linha 13) indica que `setCourseName` requer um parâmetro `string` e não retorna um valor (isto é, seu tipo de retorno é `void`). O protótipo de função da função-membro `getCourseName` (linha 14) indica que a função não requer parâmetros e retorna uma `string`.

Por fim, o protótipo da função-membro `displayMessage` (linha 15) especifica que `displayMessage` não requer parâmetros e não retorna um valor. Esses protótipos de função são os mesmos que os cabeçalhos de função correspondentes na Figura 3.9, exceto pelo fato de que os nomes de parâmetro (que são opcionais em protótipos) não são incluídos e cada protótipo de função deve terminar com um ponto-e-vírgula.



Erro comum de programação 3.8

Esquecer de colocar o ponto-e-vírgula no final de um protótipo de função é um erro de sintaxe.



Boa prática de programação 3.7

Embora nomes de parâmetro em protótipos de função sejam opcionais (eles são ignorados pelo compilador), muitos programadores utilizam esses nomes para propósitos de documentação.



Dica de prevenção de erro 3.4

Os nomes de parâmetro em um protótipo de função (que, novamente, são ignorados pelo compilador) podem ser enganosos se nomes errados ou confusos forem utilizados. Por essa razão, muitos programadores criam protótipos de função copiando a primeira linha das definições de função correspondentes (quando o código-fonte das funções estiver disponível), acrescentando então um ponto-e-vírgula ao final de cada protótipo.

```

1 // Figura 3.11: GradeBook.h
2 // Definição da classe GradeBook. Esse arquivo apresenta a interface pública de
3 // GradeBook sem revelar as implementações de funções-membro de GradeBook
4 // que são definidas em GradeBook.cpp.
5 #include <string> // a classe GradeBook utiliza a classe de string padrão C++
6 using std::string;
7
8 // definição da classe GradeBook
9 class GradeBook
10 {
11 public:
12     GradeBook( string ); // construtor que inicializa courseName
13     void setCourseName( string ); // função que configura o nome do curso
14     string getCourseName(); // função que obtém o nome do curso
15     void displayMessage(); // função que exibe uma mensagem de boas-vindas
16 private:
17     string courseName; // nome do curso para esse GradeBook
18 }; // fim da classe GradeBook

```

Figura 3.11 Definição da classe GradeBook contendo os protótipos de função que especificam a interface da classe.

```

1 // Figura 3.12: GradeBook.cpp
2 // Definições de função-membro de GradeBook. Esse arquivo contém
3 // implementações das funções-membro prototipadas em GradeBook.h.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include "GradeBook.h" // inclui a definição de classe GradeBook
9
10 // o construtor inicializa courseName com a string fornecida como argumento
11 GradeBook::GradeBook( string name )
12 {
13     setCourseName( name ); // chama a função set para inicializar courseName
14 } // fim do construtor GradeBook
15
16 // função para configurar o nome do curso
17 void GradeBook::setCourseName( string name )
18 {
19     courseName = name; // armazena o nome do curso no objeto
20 } // fim da função setCourseName
21
22 // função para obter o nome do curso
23 string GradeBook::getCourseName()
24 {
25     return courseName; // retorna courseName do objeto
26 } // fim da função getCourseName
27
28 // exibe uma mensagem de boas-vindas para o usuário GradeBook
29 void GradeBook::displayMessage()
30 {
31     // chama getCourseName para obter o courseName
32     cout << "Welcome to the grade book for\n" << getCourseName()
33         << "!" << endl;
34 } // fim da função displayMessage

```

Figura 3.12 As definições de função-membro GradeBook representam a implementação da classe GradeBook.

```

1 // Figura 3.13: fig03_13.cpp
2 // Demonstração de classe GradeBook depois de separar
3 // sua interface de sua implementação.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include "GradeBook.h" // inclui a definição de classe GradeBook
9
10 // a função main inicia a execução do programa
11 int main()
12 {
13     // cria dois objetos GradeBook
14     GradeBook gradeBook1( "CS101 Introduction to C++ Programming" );
15     GradeBook gradeBook2( "CS102 Data Structures in C++" );
16
17     // exibe valor inicial de courseName para cada GradeBook
18     cout << "gradeBook1 created for course: " << gradeBook1.getCourseName()
19         << "\ngradeBook2 created for course: " << gradeBook2.getCourseName()
20         << endl;
21     return 0; // indica terminação bem-sucedida
22 } // fim de main

```

```

gradeBook1 created for course: CS101 Introduction to C++ Programming
gradeBook2 created for course: CS102 Data Structures in C++

```

Figura 3.13 A demonstração da classe GradeBook depois de separar sua interface de sua implementação.

GradeBook.cpp: Definindo funções-membro em um arquivo de código-fonte separado

O arquivo de código-fonte GradeBook.cpp (Figura 3.12) define as funções-membro da classe GradeBook, que foram declaradas nas linhas 12–15 da Figura 3.11. As definições de função-membro aparecem nas linhas 11–34 e são quase idênticas às definições de função-membro nas linhas 15–38 da Figura 3.9.

Note que cada nome de função-membro nos cabeçalhos de função (linhas 11, 17, 23 e 29) é precedido pelo nome de classe e por ::, que é conhecido como o **operador de resolução de escopo binário**. Isso ‘amarra’ cada função-membro à definição de classe GradeBook (agora separada), que declara as funções-membro e membros de dados da classe. Sem ‘GradeBook::’ precedendo cada nome de função, essas funções não seriam reconhecidas pelo compilador como funções-membro da classe GradeBook — o compilador as consideraria funções ‘livres’ ou ‘irrestritas’, como main. Essas funções não podem acessar os dados private de GradeBook ou chamar as funções-membro da classe, sem especificar um objeto. Portanto, o compilador não seria capaz de compilar essas funções. Por exemplo, as linhas 19 e 25 que acessam a variável courseName causariam erros de compilação porque courseName não é declarada como uma variável local em cada função — o compilador não saberia que courseName já foi declarada como um membro de dados da classe GradeBook.



Erro comum de programação 3.9

Ao definir funções-membro de uma classe fora dessa classe, omitir o nome de classe e operador de resolução de escopo binário (::) que precede os nomes de função causa erros de compilação.

Para indicar que as funções-membro em GradeBook.cpp fazem parte da classe GradeBook, devemos primeiro incluir o arquivo de cabeçalho GradeBook.h (linha 8 da Figura 3.12). Isso permite acessar o nome da classe GradeBook no arquivo GradeBook.cpp. Ao compilar GradeBook.cpp, o compilador utiliza as informações em GradeBook.h para assegurar que:

1. a primeira linha de cada função-membro (linhas 11, 17, 23 e 29) corresponda a seu protótipo no arquivo GradeBook.h — por exemplo, o compilador assegura que getCourseName não aceita parâmetros e retorna uma string.
2. cada função-membro conheça os membros de dados e outras funções-membro da classe — por exemplo, as linhas 19 e 25 podem acessar a variável courseName porque ela foi declarada em GradeBook.h como um membro de dados da classe GradeBook, e as linhas 13 e 32 podem chamar as funções setCourseName e getCourseName, respectivamente, porque cada uma foi declarada como uma função-membro da classe em GradeBook.h (e porque essas chamadas obedecem aos protótipos correspondentes).

Testando a classe **GradeBook**

A Figura 3.13 realiza as mesmas manipulações de objeto de `GradeBook` que a Figura 3.10. Separar a interface de `GradeBook` da implementação de suas funções-membro não afeta a maneira como esse código-cliente utiliza a classe. Afeta apenas a maneira como o programa é compilado e vinculado, o que discutiremos detalhadamente em breve.

Como na Figura 3.10, a linha 8 da Figura 3.13 inclui o arquivo de cabeçalho `GradeBook.h` para que o compilador possa assegurar que os objetos de `GradeBook` sejam criados e manipulados corretamente no código-cliente. Antes de executar esse programa, os arquivos de código-fonte nas figuras 3.12 e 3.13 devem ser compilados, e, então, vinculados entre si — isto é, as chamadas de função-membro no código-cliente precisam ser associadas às implementações das funções-membro da classe — um trabalho realizado pelo linker.

O processo de compilação e vinculação

O diagrama na Figura 3.14 mostra o processo de compilação e vinculação que resulta de um aplicativo `GradeBook` executável que pode ser utilizado por instrutores. Frequentemente, a interface e a implementação de uma classe serão criadas e compiladas por um programador e utilizadas por um programador separado que implementa o código-cliente da classe. Então, o diagrama mostra o que é requerido tanto pelo programador de implementação da classe como pelo programador de código-cliente. As linhas tracejadas no diagrama mostram as partes requeridas pelo programador de implementação da classe, pelo programador de código-cliente e pelo usuário de aplicativo `GradeBook`, respectivamente. [Nota: A Figura 3.14 não é um diagrama UML.]

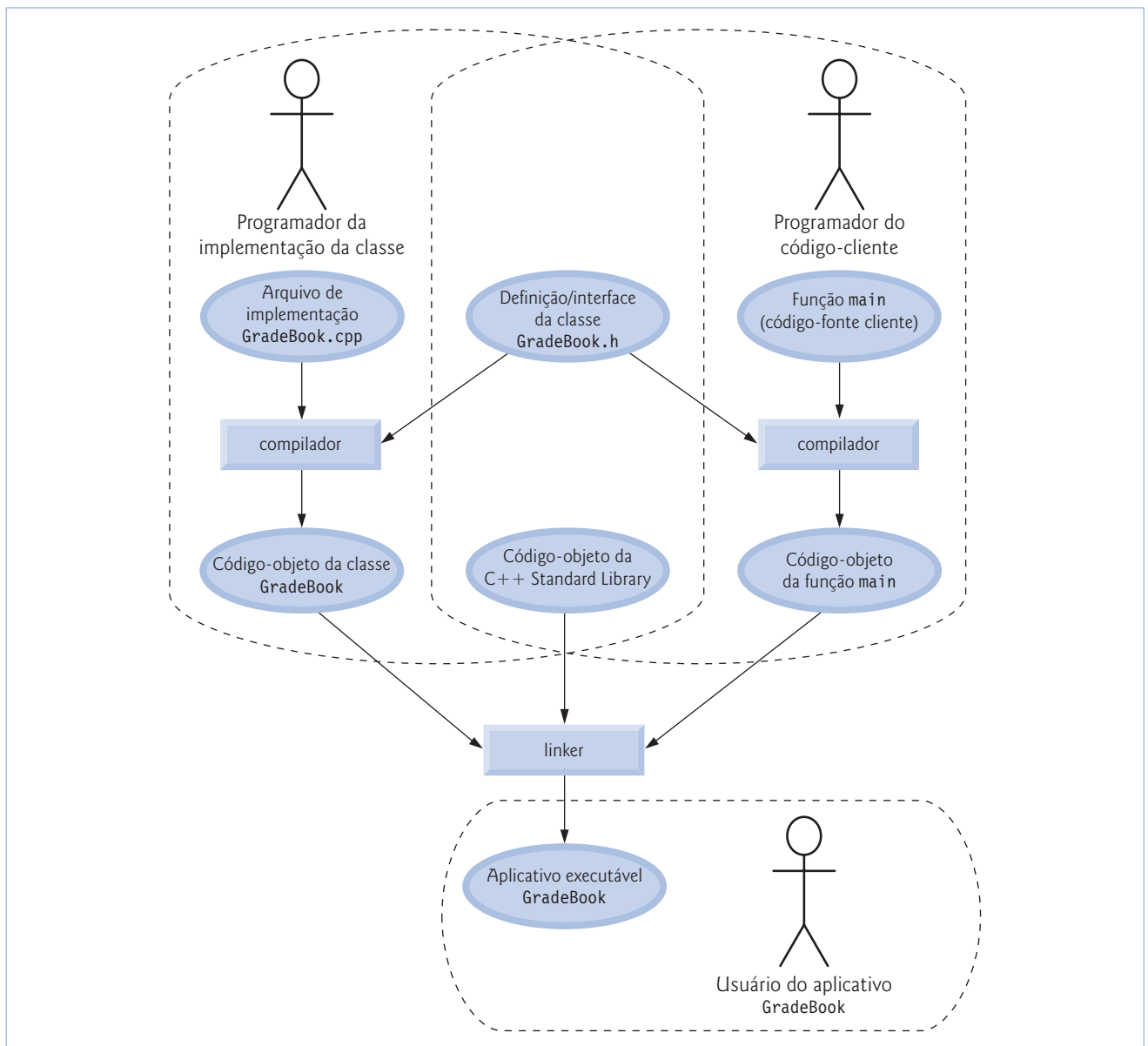


Figura 3.14 O processo de compilação e vinculação que produz um aplicativo executável.

Um programador de implementação de classe responsável por criar uma classe `GradeBook` reutilizável cria o arquivo de cabeçalho `GradeBook.h` e o arquivo de código-fonte `GradeBook.cpp` que inclui (`#include`) o arquivo de cabeçalho, e, depois, compila o arquivo de código-fonte para criar código-objeto de `GradeBook`. Para ocultar os detalhes de implementação de funções-membro de `GradeBook`, o programador da implementação da classe forneceria ao programador do código-cliente o arquivo de cabeçalho `GradeBook.h` (que especifica a interface e os membros de dados da classe) e o código-objeto para classe `GradeBook` que contém as instruções de linguagem de máquina que representam as funções-membro de `GradeBook`. O programador do código-cliente não recebe o arquivo de código-fonte `GradeBook`; portanto, o cliente permanece sem saber como as funções-membro de `GradeBook` são implementadas.

O código-cliente só precisa conhecer a interface de `GradeBook` para utilizar a classe e deve ser capaz de vincular seu código-objeto. Visto que a interface da classe faz parte da definição de classe no arquivo de cabeçalho `GradeBook.h`, o programador do código-cliente deve ter acesso a esse arquivo e incluí-lo, usando `#include`, no arquivo de código-fonte do cliente. Quando o código-cliente é compilado, o compilador utiliza a definição de classe em `GradeBook.h` para assegurar que a função `main` cria e manipula objetos da classe `GradeBook` corretamente.

Para criar o aplicativo `GradeBook` executável para ser utilizado por instrutores, o último passo é vincular

1. o código-objeto da função `main` (isto é, o código-cliente);
2. o código-objeto para implementações de função-membro da classe `GradeBook`;
3. o código-objeto da C++ Standard Library para as classes C++ (por exemplo, `string`) utilizadas pelo programador da implementação da classe e pelo programador do código-cliente.

A saída do linker é o aplicativo `GradeBook` executável que instrutores podem utilizar para gerenciar as notas de seus alunos.

Para obter informações adicionais sobre como compilar programas de múltiplos arquivos de código-fonte, veja a documentação do seu compilador ou estude as publicações *DIVE-INTO™* que oferecemos para vários compiladores C++ em www.deitel.com/books/cpphttp5.

3.10 Validando dados com funções *set*

Na Seção 3.6, introduzimos as funções *set* para permitir aos clientes de uma classe modificar o valor de um membro de dados `private`. Na Figura 3.5, a classe `GradeBook` define a função-membro `setCourseName` para simplesmente atribuir um valor recebido em seu parâmetro `name` ao membro de dados `courseName`. Essa função-membro não assegura que o nome do curso obedeça a qualquer formato particular ou siga quaisquer outras regras relacionadas ao que é um nome do curso ‘válido’. Como declaramos anteriormente, suponha que uma universidade pode imprimir listagens de alunos contendo nomes de curso com apenas 25 caracteres ou menos. Se a universidade utiliza um sistema contendo objetos `GradeBook` para gerar as listagens, poderíamos querer que a classe `GradeBook` assegurasse que seu membro de dados `courseName` nunca contivesse mais do que 25 caracteres. O programa das figuras 3.15–3.17 aprimora a função-membro `setCourseName` da classe `GradeBook` para realizar essa **validação** (também conhecida como **teste de validade**).

Definição de classe *GradeBook*

Note que a definição da classe `GradeBook` (Figura 3.15) — e, portanto, sua interface — é idêntica à da Figura 3.11. Visto que a interface permanece inalterada, os clientes dessa classe não precisam ser alterados quando a definição de função-membro `setCourseName` é

```

1 // Figura 3.15: GradeBook.h
2 // Definição de classe GradeBook apresenta a interface public da
3 // classe. Definições de função-membro aparecem em GradeBook.cpp.
4 #include <string> // o programa utiliza classe de string padrão do C++
5 using std::string;
6
7 // definição da classe GradeBook
8 class GradeBook
9 {
10 public:
11     GradeBook( string ); // construtor que inicializa um objeto GradeBook
12     void setCourseName( string ); // função que configura o nome do curso
13     string getCourseName(); // função que obtém o nome do curso
14     void displayMessage(); // função que exibe uma mensagem de boas-vindas
15 private:
16     string courseName; // nome do curso para esse GradeBook
17 }; // fim da classe GradeBook

```

Figura 3.15 A definição da classe `GradeBook`.

```

1 // Figura 3.16: GradeBook.cpp
2 // Implementações das definições de função-membro de GradeBook.
3 // A função setCourseName realiza a validação.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include "GradeBook.h" // inclui a definição de classe GradeBook
9
10 // construtor inicializa courseName com String fornecido como argumento
11 GradeBook::GradeBook( string name )
12 {
13     setCourseName( name ); // valida e armazena courseName
14 } // fim do construtor GradeBook
15
16 // função que configura o nome do curso;
17 // assegura que o nome do curso tenha no máximo 25 caracteres
18 void GradeBook::setCourseName( string name )
19 {
20     if ( name.length() <= 25 ) // se o nome tiver 25 ou menos caracteres
21         courseName = name; // armazena o nome do curso no objeto
22
23     if ( name.length() > 25 ) // se o nome tiver mais de 25 caracteres
24     {
25         // configura courseName como os primeiros 25 caracteres do parâmetro name
26         courseName = name.substr( 0, 25 ); // inicia em 0, comprimento de 25
27
28         cout << "Name \"" << name << "\" exceeds maximum length (25).\n"
29              << "Limiting courseName to first 25 characters.\n" << endl;
30     } // fim do if
31 } // fim da função setCourseName
32
33 // função para obter o nome do curso
34 string GradeBook::getCourseName()
35 {
36     return courseName; // retorna courseName do objeto
37 } // fim da função getCourseName
38
39 // exibe uma mensagem de boas-vindas para o usuário GradeBook
40 void GradeBook::displayMessage()
41 {
42     // chama getCourseName para obter o courseName
43     cout << "Welcome to the grade book for\n" << getCourseName()
44          << "!" << endl;
45 } // fim da função displayMessage

```

Figura 3.16 As definições de função-membro para a classe `GradeBook` com uma função *set* que valida o comprimento do membro de dados `courseName`.

modificada. Isso permite que os clientes tirem proveito da classe `GradeBook` melhorada simplesmente vinculando o código-cliente ao código-objeto da `GradeBook` atualizada.

Validando o nome do curso com a função-membro `setCourseName` `GradeBook`

O aprimoramento para a classe `GradeBook` está na definição de `setCourseName` (Figura 3.16, linhas 18–31). A instrução `if` nas linhas 20–21 determina se o parâmetro `name` contém um nome válido de curso (isto é, uma string de 25 ou menos caracteres). Se o nome do curso for válido, a linha 21 armazenará o nome do curso no membro de dados `courseName`. Observe a expressão `name.length()` na linha 20. Essa é uma chamada de função-membro idêntica a `myGradeBook.displayMessage()`. A classe `string` da C++ Standard

Library define uma função-membro `length` que retorna o número de caracteres em um objeto `string`. O parâmetro `name` é um objeto `string`, portanto a chamada `name.length()` retorna o número de caracteres em `name`. Se esse valor é menor que ou igual a 25, `name` é válido e a linha 21 é executada.

A instrução `if` nas linhas 23–30 trata o caso em que `setCourseName` recebe um nome inválido de curso (isto é, um nome que tem mais de 25 caracteres de comprimento). Mesmo se o parâmetro `name` for muito longo, ainda queremos deixar o objeto `GradeBook` em um **estado consistente** — isto é, um estado em que o membro de dados do objeto `courseName` contenha um valor válido (isto é, uma `string` de 25 ou menos caracteres). Portanto, truncamos (isto é, encurtamos) o nome do curso especificado e atribuímos os 25 primeiros caracteres de `name` ao membro de dados `courseName` (infelizmente, isso poderia truncar horivelmente o nome do curso). A classe `string` padrão fornece a função-membro `substr` (abreviação de ‘substring’) que retorna um novo objeto `string` criado mediante a cópia de parte de um objeto `string` existente. A chamada na linha 26 (isto é, `name.substr( 0, 25 )`) passa dois inteiros (0 e 25) para a função-membro `substr` de `name`. Esses argumentos indicam a parte da `string` `name` que `substr` deve retornar. O primeiro argumento especifica a posição inicial na `string` original a partir da qual os caracteres são copiados — considera-se que o primeiro caractere em cada `string` está na posição 0. O segundo argumento especifica o número de caracteres a ser copiado. Portanto, a chamada na linha 26 retorna uma

```

1 // Figura 3.17: fig03_17.cpp
2 // Cria e manipula um objeto GradeBook; ilustra a validação.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "GradeBook.h" // inclui a definição de classe GradeBook
8
9 // a função main inicia a execução do programa
10 int main()
11 {
12     // cria dois objetos GradeBook;
13     // nome inicial de curso de gradeBook1 é muito longo
14     GradeBook gradeBook1( "CS101 Introduction to Programming in C++" );
15     GradeBook gradeBook2( "CS102 C++ Data Structures" );
16
17     // exibe courseName de cada GradeBook
18     cout << "gradeBook1's initial course name is: "
19         << gradeBook1.getCourseName()
20         << "\ngradeBook2's initial course name is: "
21         << gradeBook2.getCourseName() << endl;
22
23     // modifica courseName do myGradeBook (com uma string de comprimento válido)
24     gradeBook1.setCourseName( "CS101 C++ Programming" );
25
26     // exibe courseName de cada GradeBook
27     cout << "\ngradeBook1's course name is: "
28         << gradeBook1.getCourseName()
29         << "\ngradeBook2's course name is: "
30         << gradeBook2.getCourseName() << endl;
31     return 0; // indica terminação bem-sucedida
32 } // fim de main

```

Name "CS101 Introduction to Programming in C++" exceeds maximum length (25).
Limiting courseName to first 25 characters.

gradeBook1's initial course name is: CS101 Introduction to Pro
gradeBook2's initial course name is: CS102 C++ Data Structures

gradeBook1's course name is: CS101 C++ Programming
gradeBook2's course name is: CS102 C++ Data Structures

Figura 3.17 Criando e manipulando um objeto `GradeBook` em que o nome do curso é limitado a 25 caracteres de comprimento.

substring de 25 caracteres de `name` que inicia na posição 0 (isto é, os primeiros 25 caracteres em `name`). Por exemplo, se `name` armazena o valor "CS101 Introduction to Programming in C++", `substr` retorna "CS101 Introduction to Pro". Depois da chamada para `substr`, a linha 26 atribui a substring retornada por `substr` ao membro de dados `courseName`. Dessa maneira, a função-membro `setCourseName` assegura que a `courseName` é sempre atribuída uma string contendo 25 ou menos caracteres. Se a função-membro tem de truncar o nome do curso para torná-lo válido, as linhas 28–29 exibem uma mensagem de advertência.

Observe que a instrução `if` nas linhas 23–30 contém duas instruções de corpo — uma para configurar o `courseName` para os primeiros 25 caracteres do parâmetro `name` e uma para imprimir uma mensagem anexa para o usuário. Queremos que essas duas instruções executem quando o `name` for muito longo, então as colocamos entre chaves, `{ }`. A partir da discussão no Capítulo 2, lembre-se de que isso cria um bloco. Você aprenderá mais sobre como colocar múltiplas instruções no corpo de uma instrução de controle no Capítulo 4.

Observe que a instrução `cout` nas linhas 28–29 também poderia aparecer sem o operador de inserção de fluxo no início da segunda linha da instrução, como em:

```
cout << "Name \"" << name << "\" exceeds maximum length (25).\n"
    "Limiting courseName to first 25 characters.\n" << endl;
```

O compilador C++ combina literais string adjacentes, mesmo se eles aparecerem em linhas separadas de um programa. Portanto, na instrução anterior, o compilador C++ combinaria os literais string `"\" exceeds maximum length (25).\n"` e `"Limiting courseName to first 25 characters.\n"` em um único literal string para produzir saída idêntica à das linhas 28–29 na Figura 3.16. Esse comportamento permite imprimir strings longas dividindo-as ao longo das linhas no programa sem incluir operações de inserção de fluxo adicionais.

Testando a classe *GradeBook*

A Figura 3.17 demonstra a versão modificada da classe *GradeBook* (figuras 3.15–3.16) destacando a validação. A linha 14 cria um objeto *GradeBook* chamado `gradeBook1`. Lembre-se de que o construtor *GradeBook* chama a função-membro `setCourseName` para inicializar o membro de dados `courseName`. Nas versões anteriores da classe, o benefício de chamar `setCourseName` no construtor não era evidente. Agora, porém, o construtor tira proveito da validação fornecida por `setCourseName`. O construtor simplesmente chama `setCourseName`, em vez de duplicar seu código de validação. Quando a linha 14 da Figura 3.17 passa um nome inicial de curso de "CS101 Introduction to Programming in C++" para o construtor *GradeBook*, o construtor passa esse valor para `setCourseName`, onde ocorre a inicialização real. Como esse nome do curso contém mais que 25 caracteres, o corpo da segunda instrução `if` executa, fazendo com que `courseName` seja inicializado como o curso do nome de 25 caracteres truncado "CS101 Introduction to Pro" (a parte truncada é destacada em *itálico* na linha 14). Note que a saída na Figura 3.17 contém a mensagem de advertência gerada pelas linhas 28–29 da Figura 3.16 na função-membro `setCourseName`. A linha 15 cria outro objeto *GradeBook* chamado `gradeBook2` — o nome do curso válido passado para o construtor tem exatamente 25 caracteres.

As linhas 18–21 da Figura 3.17 exibem o nome do curso truncado de `gradeBook1` (destacamos isso em *itálico* na saída de programa) e o nome do curso de `gradeBook2`. A linha 24 chama a função-membro `setCourseName` de `gradeBook1` diretamente, para mudar o nome do curso no objeto *GradeBook* para um nome mais curto que não precise ser truncado. Então, as linhas 27–30 geram novamente a saída dos nomes de curso dos objetos *GradeBook*.

Notas adicionais sobre as funções *set*

Uma função *set* `public` como `setCourseName` deve verificar qualquer tentativa de modificação do valor de um membro de dados (por exemplo, `courseName`) para assegurar que o novo valor seja apropriado a esse item de dados. Por exemplo, uma tentativa de *configurar* (*set*) o dia do mês como 37 deve ser rejeitada, uma tentativa de *configurar* (*set*) o peso de uma pessoa como zero ou um valor negativo deve ser rejeitada, uma tentativa de *configurar* (*set*) a nota de uma prova como 185 (quando o intervalo adequado é o de zero a 100) deve ser rejeitada etc.



Observação de engenharia de software 3.6

Tornar os membros de dados `private` e controlar o acesso, especialmente acesso de gravação, para aqueles membros de dados via funções-membro `public` ajuda a assegurar a integridade de dados.



Dica de prevenção de erro 3.5

Os benefícios da integridade de dados não são automáticos simplesmente porque os membros de dados se tornaram `private` — o programador deve fornecer teste de validade apropriado e informar os erros.



Observação de engenharia de software 3.7

*As funções-membro que configuram (*set*) os valores de dados `private` devem verificar se os novos valores projetados são adequados; se não forem, as funções *set* devem colocar os membros de dados `private` em um estado apropriado.*

As funções *set* de uma classe podem retornar valores para os clientes da classe indicando que foram feitas tentativas de atribuição de dados inválidos a objetos da classe. Um cliente da classe pode testar o valor de retorno de uma função *set* para determinar se a tentativa

do cliente de modificar o objeto foi bem-sucedida e executar uma ação apropriada. No Capítulo 16, demonstramos como os clientes de uma classe podem ser notificados por meio do mecanismo de tratamento de exceções quando uma tentativa de modificar um objeto com um valor impróprio for feita. Para manter o programa das figuras 3.15–3.17 simples neste ponto inicial no livro, `setCourseName` na Figura 3.16 imprime apenas uma mensagem apropriada na tela.

3.1.1 Estudo de caso de engenharia de software: identificando as classes no documento de requisitos do ATM (opcional)

Agora começamos a projetar o sistema ATM que introduzimos no Capítulo 2. Nesta seção, identificamos as classes que são necessárias para construir o sistema ATM analisando os substantivos simples e os substantivos compostos que aparecem no documento de requisitos. Introduzimos diagramas de classes UML para modelar os relacionamentos entre essas classes. Este é um primeiro passo importante na definição da estrutura do nosso sistema.

Identificando as classes em um sistema

Iniciamos nosso processo OOD identificando as classes necessárias para construir o sistema ATM. Por fim, descrevemos essas classes utilizando diagramas de classes UML e implementamos essas classes em C++. Inicialmente, vamos revisar o documento de requisitos da Seção 2.8 e localizar substantivos e substantivos compostos para nos ajudar a identificar classes que compreendem o sistema ATM. Podemos decidir que alguns desses substantivos e substantivos compostos são atributos de outras classes no sistema. Também podemos concluir que alguns substantivos não correspondem a partes do sistema e, portanto, simplesmente não devem ser modelados. As classes adicionais podem tornar-se visíveis à medida que avançamos no processo de projeto.

A Figura 3.18 lista os substantivos e substantivos compostos encontrados no documento de requisitos. Listamos esses substantivos da esquerda para a direita na ordem em que aparecem no documento de requisitos. Listamos somente a forma singular de cada substantivo simples ou substantivo composto.

Criamos classes apenas para os substantivos simples e os substantivos compostos que têm importância no sistema ATM. Não precisamos modelar o ‘banco’ como uma classe, porque o banco não é uma parte do sistema ATM — o banco simplesmente quer que o ATM seja construído. O ‘cliente’ e o ‘usuário’ também representam entidades fora do sistema — são importantes porque interagem com o nosso sistema ATM, mas não precisamos modelá-los como classes no software ATM. Lembre-se de que modelamos um usuário ATM (isto é, um cliente de banco) como o ator no diagrama de casos de uso da Figura 2.18.

Não modelamos a ‘cédula \$ 20’ nem o ‘envelope de depósito’ como classes. Esses são objetos físicos no mundo real, mas não fazem parte do que é automatizado. Podemos representar adequadamente a presença de contas no sistema utilizando um atributo da classe que modela o dispensador de cédulas. (Atribuímos atributos a classes na Seção 4.13.) Por exemplo, o dispensador de cédulas mantém uma contagem do número de contas que ele contém. O documento de requisitos não diz nada sobre o que sistema deve fazer com os envelopes de depósito depois de recebê-los. Podemos supor que simplesmente reconhecer o recebimento de um envelope — uma operação realizada pela classe que modela a abertura para depósito — é suficiente para representar a presença de um envelope no sistema. (Atribuímos operações às classes na Seção 6.22.)

Em nosso sistema ATM simplificado parece mais apropriado representar as várias quantias de ‘dinheiro’, incluindo o ‘saldo’ de uma conta, como atributos de outras classes. Da mesma forma, os substantivos ‘número de conta’ e ‘PIN’ representam partes significativas das informações no sistema ATM. Esses são atributos importantes de uma conta bancária. Mas não exibem comportamentos. Portanto, você pode modelá-los mais apropriadamente como atributos de uma classe de conta.

Embora o documento de requisitos costume descrever uma ‘transação’ em um sentido geral, não modelamos a noção ampla de uma transação financeira nesse momento. Em vez disso, modelamos os três tipos de transações (isto é, ‘consulta de saldo’, ‘saque’ e ‘depósito’) como classes individuais. Essas classes possuem atributos específicos necessários para executar as transações que elas representam. Por

Substantivos e substantivos compostos no documento de requisitos		
banco	dinheiro / fundos	número de conta
ATM	tela	PIN
usuário	teclado	banco de dados do banco
cliente	dispensador de cédulas	consulta de saldo
transação	cédula de \$ 20 / dinheiro	retirada/saque
conta	abertura para depósito	depósito
saldo	envelope de depósito	

Figura 3.18 Substantivos e substantivos compostos no documento de requisitos.

exemplo, uma retirada precisa saber quanto dinheiro o usuário quer sacar. Uma consulta de saldo, porém, não requer dados adicionais. Além disso, as três classes de transação exibem comportamentos únicos. Uma retirada inclui liberar dinheiro para o usuário, enquanto um depósito envolve receber envelopes de depósito do usuário. [Nota: Na Seção 13.10, ‘fatoramos’ recursos comuns a todas as transações em uma classe ‘transaction’ geral utilizando os conceitos orientados a objetos de classes abstratas e herança.]

Determinamos as classes para nosso sistema com base nos substantivos e substantivos compostos restantes da Figura 3.18. Cada um deles referencia um ou mais objetos, como os seguintes:

- ATM
- tela (*screen*)
- teclado (*keypad*)
- dispensador de cédulas (*cash dispenser*)
- abertura para depósito (*deposit slot*)
- conta (*account*)
- banco de dados do banco (*bank database*)
- consulta de saldo (*balance inquiry*)
- retirada/saque (*withdrawal*)
- depósito (*deposit*)

É provável que os elementos dessa lista sejam as classes necessárias para implementar nosso sistema.

Agora podemos modelar as classes em nosso sistema com base na lista que criamos. Escrevemos os nomes de classe no processo de projeto em letras maiúsculas — uma convenção UML — assim como quando escrevemos o código C++ real que implementa nosso projeto. Se o nome de uma classe contiver mais de uma palavra, unimos as palavras e colocamos a letra inicial de cada palavra em maiúscula (por exemplo, `MultipleWordName`). Utilizando essa convenção, criamos as classes `ATM`, `Screen`, `Keypad`, `CashDispenser`, `DepositSlot`, `Account`, `BankDatabase`, `BalanceInquiry`, `Withdrawal` e `Deposit`. Construímos nosso sistema utilizando todas essas classes como blocos de construção. Entretanto, antes de iniciarmos a construção do sistema, devemos obter um melhor entendimento de como as classes se relacionam.

Modelando classes

A UML permite modelar, via **diagrama de classes**, as classes no sistema ATM e seus inter-relacionamentos. A Figura 3.19 representa a classe `ATM`. Na UML, cada classe é modelada como um retângulo com três compartimentos. O compartimento superior contém o nome da classe centralizado horizontalmente e em negrito. O compartimento do meio contém os atributos da classe. (Discutimos os atributos na Seção 4.13 e Seção 5.11.) O compartimento inferior contém as operações da classe (discutidas na Seção 6.22). Na Figura 3.19 os compartimentos do meio e inferior estão vazios, porque ainda não determinamos os atributos e as operações dessa classe.

Os diagramas de classes também mostram os relacionamentos entre as classes do sistema. A Figura 3.20 mostra como nossas classes `ATM` e `Withdrawal` se relacionam entre si. Por ora, escolhemos modelar apenas esse subconjunto de classes para simplificar. Apresentamos um diagrama de classes mais completo a seguir nesta seção. Observe que os retângulos que representam classes nesse diagrama não estão subdivididos em compartimentos. A UML permite a supressão de atributos e operações de classe dessa maneira, quando apropriado, para criar diagramas mais legíveis. Diz-se que esse diagrama é um **diagrama elidido** — um em que algumas informações, como o conteúdo do segundo e terceiro compartimentos, não são modeladas. Colocaremos informações nesses compartimentos nas seções 4.13 e 6.22.

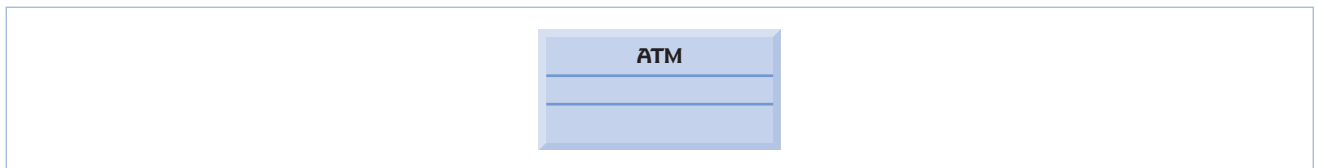


Figura 3.19 Representando uma classe na UML utilizando um diagrama de classes.

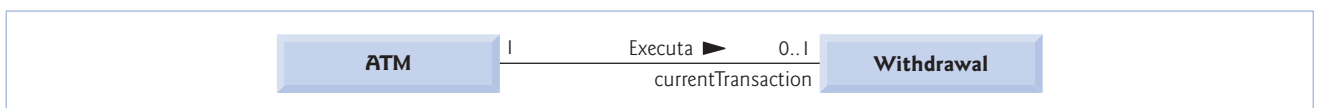


Figura 3.20 Diagrama de classes que mostra uma associação entre classes.

Na Figura 3.20, a linha sólida que conecta as duas classes representa uma **associação** — um relacionamento entre classes. Os números próximos de cada extremidade da linha são valores de **multiplicidade**, que indicam quantos objetos de cada classe participam da associação. Nesse caso, seguir a linha de uma extremidade a outra revela que, a qualquer dado momento, um objeto ATM participa de uma associação com zero ou um objeto Withdrawal — zero se o usuário atual não estiver realizando atualmente uma transação ou tiver solicitado um tipo de transação diferente, e um se o usuário tiver solicitado uma retirada. A UML pode modelar muitos tipos de multiplicidade. A Figura 3.21 lista e explica os tipos de multiplicidade.

Uma associação pode ser nomeada. Por exemplo, a palavra *Executes* acima da linha que conecta as classes ATM e Withdrawal na Figura 3.20 indica o nome dessa associação. Essa parte do diagrama exibe ‘um objeto da classe ATM que executa zero ou um objeto da classe Withdrawal’. Observe que os nomes de associação são direcionais, como indicado pela ponta da seta preenchida — então seria inadequado, por exemplo, ler a associação anterior da direita para a esquerda como ‘zero ou um objeto da classe Withdrawal que executa um objeto da classe ATM.’

A palavra *currentTransaction* ao lado de Withdrawal na linha de associação da Figura 3.20 é um **nome de papel**, que identifica o papel que o objeto Withdrawal desempenha em seu relacionamento com o ATM. Um nome de papel adiciona significado a uma associação entre classes identificando o papel que uma classe desempenha no contexto de uma associação. Uma classe pode desempenhar diversos papéis no mesmo sistema. Por exemplo, no sistema pessoal de uma escola, uma pessoa pode desempenhar o papel de ‘professor’ ao se relacionar com alunos. A mesma pessoa pode assumir o papel de ‘colega’ ao relacionar-se com outro professor e de ‘treinador’ ao treinar alunos atletas. Na Figura 3.20, o nome do papel *currentTransaction* indica que o objeto Withdrawal que participa da associação de *Executes* com um objeto da classe ATM representa a transação sendo atualmente processada pelo ATM. Em outros contextos, um objeto Withdrawal pode assumir outros papéis (por exemplo, a transação anterior). Observe que não especificamos um nome de papel para a extremidade ATM da associação *Executes*. Os nomes de papel em diagrama de classes costumam ser omitidos quando o significado de uma associação é claro sem eles.

Além de indicar relacionamentos simples, as associações podem especificar relacionamentos mais complexos, como objetos de uma classe que são compostos de objetos de outras classes. Considere um caixa automático do mundo real. Que ‘peças’ um fabricante monta para construir um ATM funcional? Nosso documento de requisitos informa que o ATM é composto de uma tela, um teclado, um dispensador de cédulas e uma abertura para depósito.

Na Figura 3.22, os **losangos sólidos** anexados às linhas de associação da classe ATM indicam que a classe ATM tem um relacionamento de **composição** com as classes Screen, Keypad, CashDispenser e DepositSlot. A composição implica um relacionamento do todo/parte. A classe que tem o símbolo de composição (o losango sólido) em sua extremidade da linha de associação é o todo (nesse caso, ATM) e as classes na extremidade das linhas de associação são as partes — nesse caso, as classes Screen, Keypad, CashDispenser e DepositSlot. As composições na Figura 3.22 indicam que um objeto da classe ATM é formado a partir de um objeto da classe Screen, um objeto da classe CashDispenser, um objeto da classe Keypad e um objeto da classe DepositSlot. O ATM ‘tem uma’ tela, um teclado, um dispensador de cédulas e uma abertura para depósito. O **relacionamento ‘tem um’** define a composição. (Veremos na seção “Estudo de caso de engenharia de software” do Capítulo 13 que o relacionamento ‘é um’ define a herança.)

De acordo com a especificação UML, os relacionamentos de composição têm as seguintes propriedades:

1. Somente uma classe no relacionamento pode representar o todo (isto é, o losango pode ser colocado somente no final da linha de associação). Por exemplo, a tela é parte do ATM ou o ATM é parte da tela, mas a tela e o ATM não podem representar o todo no relacionamento.

Símbolo	Significado
0	Nenhuma
1	Um
<i>m</i>	Um valor de inteiro
0..1	Zero ou um
<i>m, n</i>	<i>m</i> ou <i>n</i>
<i>m..n</i>	Pelo menos <i>m</i> , mas não mais do que <i>n</i>
*	Qualquer inteiro não negativo (zero ou mais)
0..*	Zero ou mais (idêntico a *)
1..*	Um ou mais

Figura 3.21 Tipos de multiplicidade.

2. As partes no relacionamento de composição só existem enquanto o todo existir, e o todo é responsável pela criação e destruição de suas partes. Por exemplo, o ato de construir um ATM inclui manufaturar suas partes. Além disso, se o ATM é destruído, sua tela, teclado, dispensador de cédulas e abertura para depósito também são destruídos.
3. Uma parte pode pertencer a somente um todo de cada vez, embora a parte possa ser removida e anexada a outro todo, o qual então assume a responsabilidade pela parte.

Os losangos sólidos em nossos diagramas de classes indicam relacionamentos de composição que satisfazem essas três propriedades. Se um relacionamento ‘tem um’ não satisfaz um ou mais desses critérios, a UML especifica que os losangos sem preenchimento são anexados às extremidades de linhas de associação para indicar **agregação** — uma forma mais fraca de composição. Por exemplo, um computador pessoal e um monitor de computador participam de um relacionamento de agregação — o computador ‘tem um’ monitor, mas as duas partes podem existir independentemente e o mesmo monitor pode ser anexado a múltiplos computadores de uma vez, violando assim a segunda e a terceira propriedades de composição.

A Figura 3.23 mostra um diagrama de classes para o sistema ATM. Esse diagrama modela a maioria das classes que identificamos anteriormente nesta seção, bem como as associações entre elas que podemos inferir do documento de requisitos. [Nota: As classes *BalanceInquiry* e *Deposit* participam de associações semelhantes àquelas da classe *Withdrawal*, então escolhemos omiti-las desse diagrama para manter o diagrama simples. No Capítulo 13, expandimos nosso diagrama de classes para incluir todas as classes no sistema ATM.]

A Figura 3.23 apresenta um modelo gráfico da estrutura do sistema ATM. Esse diagrama de classes inclui as classes *BankDatabase* e *Account* e diversas associações que não estavam presentes nas figuras 3.20 ou 3.22. O diagrama de classes mostra que a classe *ATM* tem um **relacionamento de um para um** com a classe *BankDatabase* — um objeto *ATM* autentica usuários em um objeto *BankDatabase*. Na Figura 3.23, também modelamos o fato de que o banco de dados do banco contém informações sobre muitas contas — um objeto da classe *BankDatabase* participa de um relacionamento de composição com zero ou mais objetos de classe *Account*. A partir da Figura 3.21, lembre-se de que o valor de multiplicidade $0..*$ na extremidade *Account* da associação entre a classe *BankDatabase* e a classe *Account* indica que zero ou mais objetos de classe *Account* fazem parte da associação. A classe *BankDatabase* tem um **relacionamento de um para muitos** com a classe *Account* — o *BankDatabase* armazena muitas *Accounts*. De maneira semelhante, a classe *Account* tem um **relacionamento de muitos para um** com a classe *BankDatabase* — há muitas *Accounts* armazenadas no *BankDatabase*. [Nota: A partir da Figura 3.21, lembre-se de que o valor de multiplicidade $*$ é idêntico ao $0..*$. Incluímos $0..*$ em nossos diagramas de classes para tornar isso mais claro.]

A Figura 3.23 também indica que, se o usuário estiver fazendo um saque, ‘um objeto da classe *Withdrawal* acessa/modifica o saldo de uma conta por meio de um objeto da classe *BankDatabase*’. Poderíamos ter criado uma associação direta entre a classe *Withdrawal* e a classe *Account*. O documento de requisitos, porém, determina que o ‘ATM deve interagir com o banco de dados de informações de contas do banco’ para realizar transações. Uma conta bancária contém informações sensíveis e os engenheiros de sistemas devem sempre considerar a segurança de dados pessoais ao projetar um sistema. Portanto, somente o *BankDatabase* pode acessar e manipular uma conta diretamente. Todas as outras partes do sistema devem interagir com o banco de dados para recuperar ou atualizar informações da conta (por exemplo, o saldo da conta).

O diagrama de classes na Figura 3.23 também modela associações entre a classe *Withdrawal* e as classes *Screen*, *CashDispenser* e *Keypad*. Uma transação de retirada inclui solicitar ao usuário a escolha de uma quantia em dinheiro a ser retirada e receber a entrada numérica. Essas ações requerem o uso da tela e do teclado, respectivamente. Além disso, liberar dinheiro para o usuário requer acesso ao dispensador de cédulas.

As classes *BalanceInquiry* e *Deposit*, embora não mostradas na Figura 3.23, fazem parte de diversas associações com as outras classes do sistema ATM. Como a classe *Withdrawal*, cada uma dessas classes associa-se com as classes *ATM* e *BankDatabase*. Um objeto da classe *BalanceInquiry* também se associa com um objeto da classe *Screen* para exibir o saldo de uma conta para o usuário.

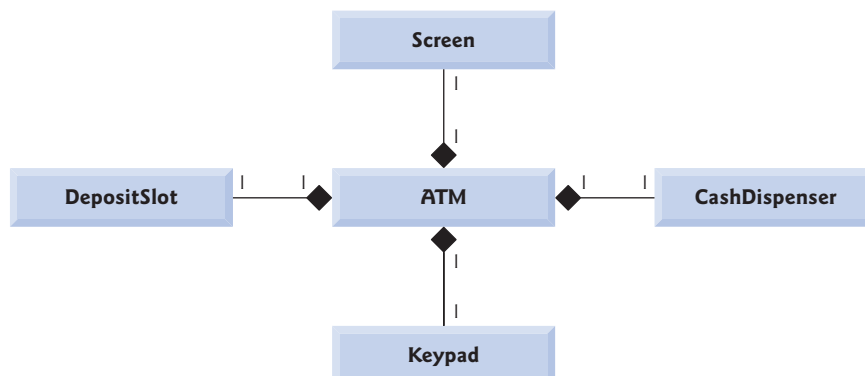


Figura 3.22 Diagrama de classes mostrando os relacionamentos de composição.

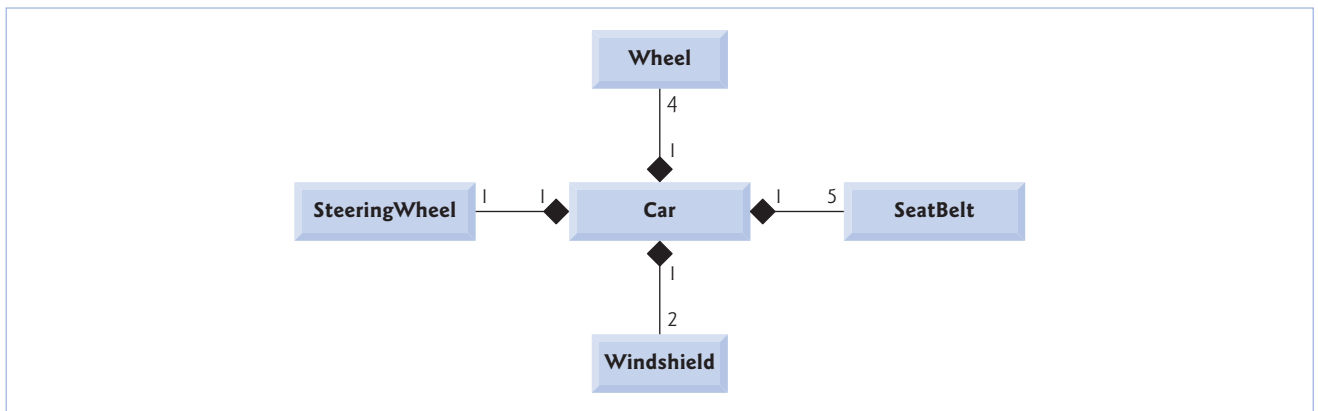


Figura 3.24 Diagrama de classes mostrando relacionamentos de composição de uma classe Car.

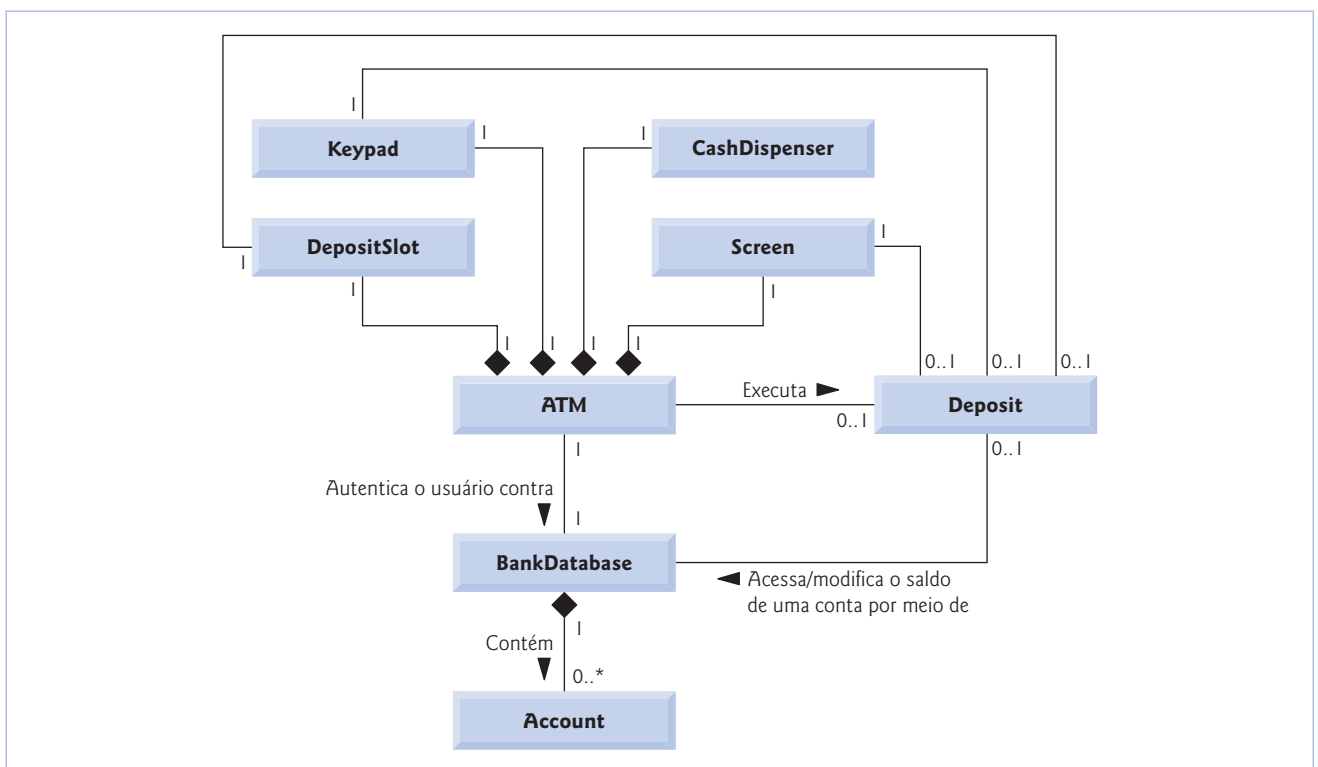


Figura 3.25 Diagrama de classes para o modelo do sistema ATM incluindo a classe Deposit.

3.12 Síntese

Neste capítulo, você aprendeu a criar classes definidas pelo usuário e a criar e utilizar objetos dessas classes. Em particular, declaramos membros de dados de uma classe para manter dados para cada objeto da classe. Definimos também as funções-membro que operam nesses dados. Você aprendeu a chamar as funções-membro de um objeto para solicitar os serviços que ele fornece e a passar dados para essas funções-membro como argumentos. Discutimos a diferença entre uma variável local de uma função-membro e um membro de dados de uma classe. Também mostramos como utilizar um construtor para especificar os valores iniciais para os membros de dados de um objeto. Você aprendeu a separar a interface de uma classe de sua implementação para promover boa engenharia de software. Apresentamos também um diagrama que mostra os arquivos que os programadores da implementação da classe e os programadores do código-cliente precisam para compilar o código que eles escrevem. Demonstramos como as funções *set* podem ser utilizadas para validar os dados de um objeto e assegurar que os objetos sejam mantidos em um estado consistente. Além disso, os diagramas de classes UML foram utilizados para modelar classes e seus construtores, funções-membro e membros de dados. No próximo capítulo iniciamos nossa introdução às instruções de controle, que especificam a ordem em que as ações de uma função são realizadas.

Resumo

- Realizar uma tarefa em um programa requer uma função. A função oculta de seu usuário as tarefas complexas que ela realiza.
- Uma função em uma classe é conhecida como uma função-membro e realiza uma das tarefas da classe.
- Você deve criar um objeto de uma classe antes de um programa realizar as tarefas que a classe descreve. Essa é uma razão pela qual C++ é conhecido como uma linguagem de programação orientada a objetos.
- Toda mensagem enviada para um objeto é uma chamada de função-membro que instrui o objeto a realizar uma tarefa.
- Um objeto tem atributos que são portados com o objeto quando ele é utilizado em um programa. Esses atributos são especificados como membros de dados na classe do objeto.
- Uma definição de classe contém os membros de dados e funções-membro que definem os atributos e os comportamentos da classe, respectivamente.
- Uma definição de classe inicia com a palavra-chave `class` seguida imediatamente pelo nome de classe.
- Por convenção, o nome de uma classe definida pelo usuário inicia com uma letra maiúscula e, por legibilidade, cada palavra subsequente no nome de classe inicia com uma letra maiúscula.
- O corpo de toda classe é incluído entre um par de chaves (`{ e }`) e termina com um ponto-e-vírgula.
- As funções-membro que aparecem depois do especificador de acesso `public` podem ser chamadas por outras funções em um programa e por funções-membro de outras classes.
- Os especificadores de acesso são sempre seguidos por dois-pontos (`:`).
- A palavra-chave `void` é um tipo de retorno especial que indica que uma função realizará uma tarefa, mas não retornará nenhum dado para sua função chamadora quando completar sua tarefa.
- Por convenção, os nomes de função-membro iniciam com a primeira letra minúscula e todas as palavras subseqüentes no nome iniciam com uma letra maiúscula.
- Um conjunto vazio de parênteses depois do nome de uma função indica que a função não requer dados adicionais para realizar sua tarefa.
- O corpo de todas as funções é delimitado por uma chave esquerda e uma chave direita (`{ e }`).
- Em geral, você não pode chamar uma função-membro até que você crie um objeto de sua classe.
- Cada nova classe que você cria torna-se um novo tipo em C++ que pode ser utilizado para declarar variáveis e criar objetos. Essa é uma razão pela qual C++ é conhecido como uma linguagem extensível.
- Uma função-membro pode requerer um ou mais parâmetros que representam dados adicionais de que ele precisa para realizar sua tarefa. Uma chamada de função fornece argumentos para cada um dos parâmetros da função.
- Uma função-membro é chamada utilizando o nome de objeto seguido por um operador ponto (`.`), pelo nome de função e por um conjunto de parênteses contendo os argumentos da função.
- Uma variável de classe `string` da C++ Standard Library representa uma string de caracteres. Essa classe é definida no arquivo de cabeçalho `<string>` e o nome `string` pertence ao namespace `std`.
- A função `getline` (do cabeçalho `<string>`) lê caracteres de seu primeiro argumento até que um caractere nova linha seja encontrado, então, coloca os caracteres (não incluindo o caractere nova linha) na variável `string` especificada como seu segundo argumento. O caractere de nova linha é descartado.
- Uma lista de parâmetros pode conter qualquer número de parâmetros, incluindo nenhum (representado por parênteses vazios) para indicar que uma função não requer parâmetros.
- O número de argumentos em uma chamada de função deve corresponder ao número de parâmetros na lista de parâmetros do cabeçalho da função-membro chamada. Além disso, os tipos de argumento na chamada de função devem ser consistentes com os tipos dos parâmetros correspondentes no cabeçalho de função.
- As variáveis declaradas no corpo de uma função são variáveis locais e só podem ser utilizadas a partir de sua declaração na função até a chave direita (`}`) imediatamente seguinte. Quando uma função termina, os valores de suas variáveis locais são perdidos.
- Uma variável local deve ser declarada antes de poder ser utilizada em uma função. Uma variável local não pode ser acessada fora da função em que é declarada.
- Normalmente, os membros de dados são `private`. As variáveis ou funções declaradas `private` são acessíveis apenas às funções-membro da classe em que elas são declaradas.
- Quando um programa cria (instancia) um objeto de uma classe, seus membros de dados `private` são encapsulados (ocultos) no objeto e podem ser acessados apenas por funções-membro da classe do objeto.
- Quando uma função que especifica um tipo de retorno diferente de `void` é chamada e completa sua tarefa, a função retorna um resultado para sua função chamadora.

- Por padrão, o valor inicial de uma `string` é a string vazia — isto é, uma string que não contém caracteres. Nada aparece na tela quando uma string vazia é exibida.
- As classes frequentemente fornecem funções-membro `public` para permitir aos clientes da classe *configurar* (*set*) ou *obter* (*get*) membros de dados `private`. Os nomes dessas funções-membro normalmente iniciam com *set* ou *get*.
- Fornecer funções públicas *set* e *get* permite aos clientes de uma classe acessar indiretamente os dados ocultos. O cliente sabe que está tentando modificar ou obter os dados de um objeto, mas não sabe como o objeto realiza essas operações.
- As funções *set* e *get* de uma classe também devem ser utilizadas por outras funções-membro dentro da classe para manipular os dados `private` da classe, embora essas funções-membro possam acessar os dados `private` diretamente. Se a representação de dados da classe é alterada, as funções-membro que acessam os dados apenas por meio das funções *set* e *get* não exigirão modificação — somente os corpos das funções *set* e *get* que manipulam diretamente o membro de dados precisarão mudar.
- Uma função *set* `public` deve verificar qualquer tentativa de modificação do valor de um membro de dados para assegurar que o novo valor seja apropriado àquele item de dados.
- Cada classe que você declara deve fornecer um construtor para inicializar um objeto da classe quando o objeto é criado. Um construtor é uma função-membro especial que deve ser definida com o mesmo nome da classe, de modo que o compilador possa diferenciá-lo de outras funções-membro da classe.
- Uma diferença entre construtores e funções é que os construtores não podem retornar valores, portanto não podem especificar um tipo de retorno (nem mesmo `void`). Normalmente, os construtores são declarados `public`.
- O C++ requer uma chamada de construtor no momento em que cada objeto é criado, o que ajuda a assegurar que cada objeto é inicializado antes de ser utilizado em um programa.
- Um construtor que não aceita argumentos é um construtor-padrão. Em qualquer classe que não inclui um construtor, o compilador fornece um construtor-padrão. O programador de classe também pode definir um construtor-padrão explicitamente. Se o programador definir um construtor para uma classe, o C++ não criará um construtor-padrão.
- As definições de classe, quando empacotadas adequadamente, podem ser reutilizadas por programadores em todo o mundo.
- É comum definir uma classe em um arquivo de cabeçalho que tenha uma extensão de nome do arquivo `.h`.
- Se a implementação da classe mudar, os clientes da classe não devem precisar mudar.
- As interfaces definem e padronizam as maneiras como coisas, pessoas e sistemas interagem.
- A interface de uma classe descreve as funções-membro `public` (também conhecidas como serviços `public`) que são disponibilizados para os clientes da classe. A interface descreve *que* serviços os clientes podem utilizar e como *solicitar* esses serviços, mas não especifica *como* a classe executa os serviços.
- Um princípio fundamental da boa engenharia de software é separar a interface da implementação. Isso torna os programas mais fáceis de modificar. Alterações na implementação da classe não afetam o cliente contanto que a interface da classe originalmente fornecida para o cliente permaneça inalterada.
- Um protótipo de função contém o nome de uma função, seu tipo de retorno e o número, tipos e a ordem dos parâmetros que a função espera receber.
- Uma vez que uma classe é definida e suas funções-membro são declaradas (via protótipos de função), as funções-membro devem ser definidas em um arquivo de código-fonte separado.
- Para cada função-membro definida fora de sua definição de classe correspondente, o nome de função deve ser precedido pelo nome de classe e pelo operador de resolução de escopo binário (`::`).
- A função-membro `length` da classe `string` retorna o número de caracteres em um objeto `string`.
- A função-membro `substr` da classe `string` (abreviação de ‘substring’) retorna um novo objeto `string` criado copiando parte de um objeto `string` existente. O primeiro argumento da função especifica a posição inicial na `string` original a partir da qual caracteres são copiados. Seu segundo argumento especifica o número de caracteres a copiar.
- Na UML, cada classe é modelada em um diagrama de classes como um retângulo com três compartimentos. O compartimento superior contém o nome de classe, centralizado horizontalmente em negrito. O compartimento do meio contém os atributos da classe (membros de dados em C++). O compartimento inferior contém as operações da classe (funções-membro e construtores em C++).
- A UML modela operações listando o nome da operação seguido por um conjunto de parênteses. Um sinal de adição (+) que precede o nome de operação indica uma operação `public` na UML (isto é, uma função-membro `public` em C++).
- A UML modela um parâmetro de uma operação listando o nome do parâmetro, seguido por um caractere de dois-pontos e o tipo de parâmetro entre os parênteses depois do nome de operação.
- A UML tem seus próprios tipos de dados. Nem todos os tipos de dados UML têm os mesmos nomes que os tipos C++ correspondentes. O tipo UML `String` corresponde ao tipo C++ `string`.
- A UML representa os membros de dados como atributos listando o nome do atributo, seguido por um caractere de dois-pontos e o tipo de atributo. Os atributos privados são precedidos por um sinal de subtração (–) na UML.

- A UML indica o tipo de retorno de uma operação colocando dois-pontos e o tipo de retorno depois dos parênteses que se seguem ao nome da operação.
- Os diagramas de classes UML não especificam tipos de retorno para operações que não retornam valores.
- A UML modela os construtores como operações em um terceiro compartimento do diagrama de classes. Para distinguir entre um construtor e operações de uma classe, a UML coloca a palavra 'constructor' entre aspas francesas (« e ») antes do nome do construtor.

Terminologia

argumento	grafia camelo
arquivo de cabeçalho	implementação de uma classe
arquivo de cabeçalho <string>	instância de uma classe
arquivo de código-fonte	interface de uma classe
aspas francesas, « e » (UML)	invocar uma função-membro
atributo (UML)	linguagem extensível
cabeçalho de função	lista de parâmetro
chamada de função	membro de dados
chamada de função-membro	mensagem (envio a um objeto)
cliente de um objeto ou classe	ocultamento de dados
código-objeto	operação (UML)
compartimento em um diagrama de classes (UML)	operador de resolução de escopo binário (: :)
constructor	operador ponto (.)
construtor-padrão	parâmetro
corpo de uma definição de classe	parâmetro de operação (UML)
definição de classe	precisão
definir uma classe	precisão-padrão
diagrama de classes (UML)	programador de código-cliente
engenharia de software	programador de implementação de classe
especificador de acesso	protótipo de função
especificador de acesso private	separar interface da implementação
especificador de acesso public	serviços public de uma classe
estado consistente	sinal (-) de subtração (UML)
função chamadora (chamador)	sinal (+) de adição (UML)
função de acesso	string vazia
função <i>get</i>	string, classe
função <i>getline</i> da biblioteca <string>	tipo de retorno
função modificadora	validação
função <i>set</i>	validade, verificação
função-membro	variável local
função-membro <i>length</i> da classe <i>string</i>	void, tipo de retorno
função-membro <i>substr</i> da classe <i>string</i>	

Exercícios de revisão

3.1 Preencha as lacunas em cada uma das seguintes sentenças:

- Uma casa está para uma planta arquitetônica assim como um(a) _____ está para uma classe.
- Toda definição de classe contém a palavra-chave _____ seguida imediatamente do nome da classe.
- Em geral, uma definição de classe é armazenada em um arquivo com a extensão de nome do arquivo _____.
- Cada parâmetro em um cabeçalho de função deve especificar tanto um(a) _____ como um(a) _____.
- Quando cada objeto de uma classe mantém sua própria cópia de um atributo, a variável que representa o atributo também é conhecida como um(a) _____.
- A palavra-chave *public* é um(a) _____.
- O tipo de retorno _____ indica que uma função realizará uma tarefa, mas não retornará nenhuma informação quando completar sua tarefa.
- A função _____ da biblioteca <string> lê caracteres até um caractere nova linha ser encontrado, então copia esses caracteres para a *string* especificada.

- i) Quando uma função-membro é definida fora da definição de classe, o cabeçalho de função deve incluir o nome de classe e o(a) _____, seguido(a) pelo nome de função para 'associar' a função-membro à definição de classe.
- j) O arquivo de código-fonte e quaisquer outros arquivos que utilizam uma classe podem incluir o arquivo de cabeçalho da classe via uma diretiva de pré-processador _____.

3.2 Determine se cada uma das seguintes sentenças é verdadeira ou falsa. Se falsa, explique por quê.

- a) Por convenção, os nomes de função iniciam com uma letra maiúscula e todas as palavras subsequentes no nome iniciam com uma letra maiúscula.
- b) Os parênteses vazios depois de um nome de função em um protótipo de função indicam que a função não requer parâmetros para realizar sua tarefa.
- c) Os membros de dados ou funções-membro declarados com o especificador de acesso `private` são acessíveis a funções-membro da classe em que eles são declarados.
- d) As variáveis declaradas no corpo de uma função-membro particular são conhecidas como membros de dados e podem ser utilizadas em todas as funções-membro da classe.
- e) O corpo de todas as funções é delimitado por uma chave esquerda e uma chave direita (`{` e `}`).
- f) Qualquer arquivo de código-fonte que contenha `int main()` pode ser utilizado para executar um programa.
- g) Os tipos de argumentos em uma chamada de função devem corresponder aos tipos dos parâmetros correspondentes na lista de parâmetros do protótipo de função.

3.3 Qual a diferença entre uma variável local e um membro de dados?

3.4 Explique o propósito de um parâmetro de função. Qual a diferença entre um parâmetro e um argumento?

Respostas dos exercícios de revisão

3.1 a) objeto. b) `class`. c) `.h` d) tipo, nome. e) membro de dados. f) especificador de acesso. g) `void`. h) `getline`. i) operador de resolução de escopo binário (`::`). j) `#include`.

3.2 a) Falsa. Por convenção, os nomes de função iniciam com uma letra minúscula e todas as palavras subsequentes no nome iniciam com uma letra maiúscula. b) Verdadeira. c) Verdadeira. d) Falsa. Essas variáveis são chamadas variáveis locais e só podem ser utilizadas nas funções-membro em que são declaradas. e) Verdadeira. f) Verdadeira. g) Verdadeira.

3.3 A variável local é declarada no corpo de uma função e só pode ser utilizada a partir do ponto em que é declarada até a chave direita imediatamente seguinte. Um membro de dados é declarado em uma definição de classe, mas não no corpo de qualquer das funções-membro da classe. Todo objeto (instância) de uma classe tem uma cópia separada dos membros de dados da classe. Além disso, os membros de dados são acessíveis a todas as funções-membro da classe.

3.4 Um parâmetro representa informações adicionais que uma função requer para realizar sua tarefa. Cada parâmetro requerido por uma função é especificado no cabeçalho de função. Um argumento é o valor fornecido na chamada de função. Quando a função é chamada, o valor de argumento é passado no parâmetro de função para que a função possa realizar sua tarefa.

Exercícios

3.5 Explique a diferença entre um protótipo de função e uma definição de função.

3.6 O que é um construtor-padrão? Como os membros de dados de um objeto são inicializados se uma classe tiver apenas um construtor-padrão implicitamente definido?

3.7 Explique o propósito de um membro de dados.

3.8 O que é um arquivo de cabeçalho? O que é um arquivo de código-fonte? Discuta o propósito de cada.

3.9 Explique como um programa poderia utilizar a classe `string` sem inserir uma declaração `using`.

3.10 Explique por que uma classe poderia fornecer uma função `set` e uma função `get` para um membro de dados.

3.11 (Modificando a classe `GradeBook`) Modifique a classe `GradeBook` (figuras 3.11–3.12) como mostrado a seguir:

- a) Inclua um segundo membro de dados `string` que representa o nome do instrutor de curso.
 - b) Forneça uma função `set` para alterar o nome do instrutor e uma função `get` para recuperá-lo.
 - c) Modifique o construtor para especificar dois parâmetros — um para o nome do curso e um para o nome do instrutor.
 - d) Modifique a função-membro `displayMessage` de tal maneira que ele primeiro gere a saída da mensagem de boas-vindas e o nome do curso, depois gere a saída de "This course is presented by: " seguido pelo nome do instrutor.
- Utilize sua classe modificada em um programa de teste que demonstra as novas capacidades da classe.

3.12 (Classe `Account`) Crie uma classe chamada `Account` que um banco poderia utilizar para representar contas bancárias dos clientes. Sua classe deve incluir um membro de dados de tipo `int` para representar o saldo da conta. [Nota: Nos capítulos subsequentes, utilizaremos números que contêm pontos de fração decimal (por exemplo, 2,75) — chamados valores de ponto flutuante — para representar quantias

em dólar.] Sua classe deve fornecer um construtor que recebe um saldo inicial e o utiliza para inicializar o membro de dados. O construtor deve validar o saldo inicial para assegurar que ele seja maior que ou igual a 0. Se não, o saldo deve ser configurado como 0 e o construtor deve exibir uma mensagem de erro, indicando que o saldo inicial era inválido. A classe deve fornecer três funções-membro. A função-membro `credit` deve adicionar uma quantia ao saldo atual. A função-membro `debit` deve retirar o dinheiro de `Account` e assegurar que a quantia de débito não exceda o saldo de `Account`. Se exceder, o saldo deve permanecer inalterado e a função deve imprimir uma mensagem que indica "Debit amount exceeded account balance." A função-membro `getBalance` deve retornar o saldo atual. Crie um programa que cria dois objetos `Account` e testa as funções-membro da classe `Account`.

- 3.13** (*Classe Invoice*) Crie uma classe chamada `Invoice` que uma loja de suprimentos de informática possa utilizar para representar uma fatura de um item vendido na loja. Uma `Invoice` (fatura) deve incluir quatro partes das informações como membros de dados — um número identificador (tipo `string`), uma descrição (tipo `string`), a quantidade comprada de um item (tipo `int`) e o preço por item (tipo `int`). [Nota: Nos capítulos subsequentes, utilizaremos números que contêm pontos de fração decimal (por exemplo, 2,75) — chamados valores de ponto flutuante — para representar quantias em dólar.] Sua classe deve ter um construtor que inicializa os quatro membros de dados. Forneça uma função `set` e uma função `get` para cada membro de dados. Além disso, forneça uma função-membro chamada `getInvoiceAmount` que calcula a quantia da fatura (isto é, multiplica a quantidade pelo preço por item) e depois retorna a quantidade como um valor `int`. Se a quantidade não for positiva, ela deve ser configurada como 0. Se o preço por item não for positivo, ele deve ser configurado como 0. Escreva um programa de teste que demonstre as capacidades da classe `Invoice`.
- 3.14** (*Classe Employee*) Crie uma classe chamada `Employee` que inclua três partes de informações como membros de dados — um nome (tipo `string`), um sobrenome (tipo `string`) e um salário mensal (tipo `int`). [Nota: Nos capítulos subsequentes, utilizaremos números que contêm pontos de fração decimal (por exemplo, 2,75) — chamados valores de ponto flutuante — para representar quantias em dólar.] Sua classe deve ter um construtor que inicialize os três membros de dados. Forneça uma função `set` e uma função `get` para cada membro de dados. Se o salário mensal não for positivo, configure-o como 0. Escreva um programa de teste que demonstre as capacidades da classe `Employee`. Crie dois objetos `Employee` e exiba o salário *anual* de cada objeto. Então dê a cada `Employee` um aumento de 10% e exiba novamente o salário anual de cada `Employee`.
- 3.15** (*Classe Date*) Crie uma classe chamada `Date` que inclua três partes de informações como membros de dados — um mês (tipo `int`), um dia (tipo `int`) e um ano (tipo `int`). Sua classe deve ter um construtor com três parâmetros que utilize os parâmetros para inicializar os três membros de dados. Para o propósito desse exercício, assuma que os valores fornecidos para o ano e o dia são corretos, mas certifique-se de que o valor de mês esteja no intervalo 1–12; se não estiver, configure o mês como 1. Forneça uma função `set` e uma função `get` para cada membro de dados. Forneça uma função-membro `displayDate` que exiba o dia, o mês e o ano separados por barras normais (/). Escreva um programa de teste que demonstre as capacidades da classe `Date`.



Vamos todos dar um passo para a frente.

Lewis Carroll

A roda já deu uma volta completa.

William Shakespeare

Quantas maçãs não caíram na cabeça de Newton antes de ele ter percebido a dica!

Robert Frost

Toda a evolução que conhecemos procede do vago para o definido.

Charles Sanders Peirce

Instruções de controle: parte I

OBJETIVOS

Neste capítulo, você aprenderá:

- Técnicas básicas para solução de problemas.
- A desenvolver algoritmos por meio do processo de refinamento passo a passo de cima para baixo.
- A utilizar as instruções de seleção `if` e `if...else` para escolher entre ações alternativas.
- Como utilizar a instrução de repetição `while` para executar instruções em um programa repetidamente.
- Repetição controlada por contador e repetição controlada por sentinela.
- Como utilizar os operadores de incremento, decremento e atribuição.

- 4.1 Introdução
- 4.2 Algoritmos
- 4.3 Pseudocódigo
- 4.4 Estruturas de controle
- 4.5 Instrução de seleção `if`
- 4.6 A instrução de seleção dupla `if...else`
- 4.7 A instrução de repetição `while`
- 4.8 Formulando algoritmos: repetição controlada por contador
- 4.9 Formulando algoritmos: repetição controlada por sentinela
- 4.10 Formulando algoritmos: instruções de controle aninhadas
- 4.11 Operadores de atribuição
- 4.12 Operadores de incremento e decremento
- 4.13 Estudo de caso de engenharia de software: identificando atributos de classe no sistema ATM (opcional)
- 4.14 Síntese

Resumo | Terminologia | Exercícios de revisão | Respostas dos exercícios de revisão | Exercícios

4.1 Introdução

Antes de escrever um programa para resolver um problema, devemos ter um entendimento completo do problema e uma abordagem cuidadosamente planejada para resolvê-lo. Ao escrever um programa, também devemos entender os tipos de blocos de construção que estão disponíveis e empregar técnicas comprovadas de construção de programa. Neste capítulo e no Capítulo 5, “Instruções de controle: parte 2”, discutimos essas questões na apresentação da teoria e princípios da programação estruturada. Os conceitos apresentados aqui são cruciais para construir classes eficientes e manipular objetos.

Neste capítulo introduzimos as instruções `if`, `if...else` e `while` do C++, três dos blocos de construção que permitem aos programadores especificar a lógica requerida para que funções-membro realizem suas tarefas. Dedicamos uma parte deste capítulo (e os capítulos 5 e 7) para desenvolver em mais detalhes a classe `GradeBook` introduzida no Capítulo 3. Em particular, adicionamos uma função-membro à classe `GradeBook` que utiliza as instruções de controle para calcular a média de um conjunto de notas de alunos. Outro exemplo demonstra maneiras adicionais de combinar instruções de controle para resolver um problema semelhante. Introduzimos os operadores de atribuição do C++ e exploramos os operadores de incremento e decremento do C++. Esses operadores adicionais abreviam e simplificam muitas instruções de programa.

4.2 Algoritmos

Qualquer problema de computação solucionável pode ser resolvido pela execução de uma série de ações em uma ordem específica. Um **procedimento** para resolver um problema em termos de

1. **ações** a executar e
2. **ordem** em que essas ações executam

é chamado **algoritmo**. O exemplo a seguir demonstra que é importante especificar corretamente a ordem em que as ações executam.

Considere o ‘algoritmo cresça e brilhe’ seguido por um executivo júnior para sair da cama e ir trabalhar: (1) levantar-se da cama, (2) tirar o pijama, (3) tomar banho, (4) vestir-se, (5) tomar o café da manhã, (6) dirigir o carro até o trabalho. Essa rotina leva o executivo a trabalhar bem preparado para tomar decisões críticas. Suponha que os mesmos passos sejam seguidos em uma ordem um pouco diferente: (1) levantar-se de cama, (2) tirar o pijama, (3) vestir-se, (4) tomar banho, (5) tomar o café da manhã, (6) dirigir o carro até o trabalho. Nesse caso, nosso executivo júnior aparece molhado no trabalho. Especificar a ordem em que as instruções (ações) são executadas em um programa de computador é chamado **controle do programa**. Este capítulo investiga o controle de programa utilizando as **instruções de controle** do C++.

4.3 Pseudocódigo

Pseudocódigo (ou código ‘fictício’) é uma linguagem artificial e informal que ajuda os programadores a desenvolver algoritmos sem a preocupação com os rigorosos detalhes da sintaxe de linguagem C++. O pseudocódigo que apresentamos aqui é particularmente útil para desenvolver algoritmos que serão convertidos em partes estruturadas de programas C++. O pseudocódigo é similar à língua cotidiana; é conveniente e amigável ao usuário embora não seja uma linguagem de programação de computador real.

O pseudocódigo não é executado nos computadores. Mais exatamente, ele ajuda o programador a ‘estudar’ um programa antes de tentar escrevê-lo em uma linguagem de programação como C++. Este capítulo fornece vários exemplos de como utilizar o pseudocódigo para desenvolver programas C++.

O estilo do pseudocódigo que apresentamos consiste puramente em caracteres, de modo que os programadores possam digitar o pseudocódigo convenientemente, utilizando um programa editor qualquer. O computador pode produzir uma nova cópia impressa de um programa em pseudocódigo sob demanda. Um programa em pseudocódigo cuidadosamente preparado pode ser facilmente convertido em um programa C++ correspondente. Em muitos casos, isso simplesmente requer a substituição de instruções em pseudocódigo por equivalentes em C++.

O pseudocódigo normalmente descreve apenas **instruções executáveis**, que fazem com que ações específicas ocorram depois que um programador converte um programa do pseudocódigo em C++ e o programa é executado em um computador. As declarações (que não têm inicializadores ou não envolvem chamadas de construtor) não são instruções executáveis. Por exemplo, a declaração

```
int i;
```

informa ao compilador o tipo da variável `i` e o instrui a reservar espaço na memória para a variável. Essa declaração não faz com que qualquer ação — como entrada, saída ou cálculo — ocorra quando o programa é executado. Em geral, não incluímos declarações de variáveis no nosso pseudocódigo. Entretanto, alguns programadores optam por listar as variáveis e mencionar seus propósitos no começo dos programas em pseudocódigo.

Agora examinamos um exemplo em pseudocódigo que pode ser escrito para ajudar um programador a criar o programa de adição da Figura 2.5. Esse pseudocódigo (Figura 4.1) corresponde ao algoritmo que insere dois inteiros a partir do usuário, adiciona esses inteiros e exibe sua soma. Embora mostremos a listagem completa em pseudocódigo aqui, mostraremos como criar pseudocódigo a partir da declaração de um problema mais adiante no capítulo.

As linhas 1–2 correspondem às instruções nas linhas 13–14 da Figura 2.5. Note que as instruções em pseudocódigo são simplesmente instruções em linguagem natural que expressam qual é a tarefa a ser realizada em C++. Da mesma forma, as linhas 4–5 correspondem às instruções nas linhas 16–17 da Figura 2.5 e as linhas 7–8 correspondem às instruções nas linhas 19 e 21 da Figura 2.5.

Há alguns aspectos importantes do pseudocódigo na Figura 4.1. Note que o pseudocódigo corresponde ao código somente na função `main`. Isso ocorre porque o pseudocódigo é normalmente utilizado para algoritmos, não para programas completos. Nesse caso, o pseudocódigo é utilizado para representar o algoritmo. A função em que esse código é colocado não é importante para o algoritmo em si. Pela mesma razão, a linha 23 da Figura 2.5 (a instrução `return`) não é incluída no pseudocódigo — essa instrução `return` é colocada no final de cada função `main` e não é importante para o algoritmo. Por fim, as linhas 9–11 da Figura 2.5 não são incluídas no pseudocódigo porque essas declarações de variável não são instruções executáveis.

4.4 Estruturas de controle

Normalmente, instruções em um programa são executadas uma após a outra na ordem em que são escritas. Isso é chamado **execução sequencial**. Várias instruções C++ que discutiremos em breve permitem ao programador especificar que a próxima instrução a executar pode ser diferente da próxima instrução na sequência. Isso é chamado **transferência de controle**.

Durante a década de 1960 tornou-se claro que a utilização indiscriminada de transferências de controle era a raiz de muitas dificuldades experimentadas por grupos de desenvolvimento de software. A culpada disso é a **instrução goto**, que permite ao programador especificar uma transferência de controle para um de vários possíveis destinos em um programa (criando o que é freqüentemente chamado de ‘código espagete’). A noção da chamada **programação estruturada** tornou-se quase sinônimo de ‘eliminação do goto’.

A pesquisa de Böhm e Jacopini¹ demonstrou que poderiam ser escritos programas sem nenhuma instrução `goto`. Tornou-se o desafio da era para programadores mudar seus estilos para ‘programação sem goto’. Mas foi só na década de 1970 que os programadores começaram a tratar programação estruturada seriamente. Os resultados foram impressionantes, pois grupos de desenvolvimento de software informaram que tinham reduzido o tempo de desenvolvimento, que a entrega de sistemas ocorria com mais freqüência dentro do prazo e

- 1 Solicite que o usuário insira o primeiro inteiro
- 2 Insira o primeiro inteiro
- 3
- 4 Solicite que o usuário insira o segundo inteiro
- 5 Insira o segundo inteiro
- 6
- 7 Some o primeiro e o segundo inteiros, armazene o resultado
- 8 Exiba o resultado

Figura 4.1 Pseudocódigo para o programa de adição da Figura 2.5.

¹ Böhm, C., & G. Jacopini, “Flow diagrams, turing machines, and languages with only two formation rules”, *Communications of the ACM*, Vol. 9, No. 5, maio de 1966, p. 366–371.

que a conclusão do projeto de software ocorria com mais frequência dentro do orçamento. A chave para esses sucessos é que programas estruturados são mais claros, mais fáceis de depurar, testar e modificar e menos propensos a conter bugs.

O trabalho de Böhm e Jacopini demonstrou que todos os programas poderiam ser escritos em termos de somente três **estruturas de controle**, a saber, a **estrutura de sequência**, a **estrutura de seleção** e a **estrutura de repetição**. O termo ‘estruturas de controle’ vem do campo da ciência da computação. Quando introduzirmos as implementações de estruturas de controle do C++, iremos nos referir a eles seguindo a terminologia do documento-padrão do C++² como ‘instruções de controle’.

Estrutura de sequência em C++

A estrutura de sequência é construída no C++. A menos que instruído de outra maneira, o computador executa as instruções C++ uma depois da outra na ordem em que elas são escritas — isto é, em sequência. O **diagrama de atividades** Unified Modeling Language (UML) da Figura 4.2 ilustra uma típica estrutura de sequência em que dois cálculos são realizados na ordem. O C++ permite ter quantas ações quisermos em uma estrutura de sequência. Como veremos logo, uma única ação pode ser colocada em qualquer lugar do código e também podemos colocar várias ações em sequência.

Nessa figura, as duas instruções envolvem adicionar uma nota a uma variável `total` e adicionar o valor 1 a uma variável `counter`. Tais instruções podem aparecer em um programa que aceita a média das notas de vários alunos. Para calcular uma média, o total das notas cuja média está sendo calculada é dividido pelo número de notas. Uma variável contadora seria utilizada para monitorar o número de valores cuja média está sendo calculada. Você verá instruções semelhantes no programa da Seção 4.8.

Os diagramas de atividades são parte da UML. Um diagrama de atividades modela o **fluxo de trabalho** (também chamado **atividade**) de uma parte de um sistema de software. Esses fluxos de trabalho podem incluir uma parte de um algoritmo, como a estrutura de sequência na Figura 4.2. Os diagramas de atividades são compostos de símbolos de uso especial, como **símbolos de estado de ação** (um retângulo com seus lados esquerdo e direito substituídos por arcos que se curvam para fora), **losangos** e **círculos pequenos**; esses símbolos são conectados por **setas de transição**, que representam o fluxo da atividade.

Como ocorre com o pseudocódigo, diagramas de atividades ajudam os programadores a desenvolver e representar algoritmos, embora muitos programadores prefiram o pseudocódigo. Os diagramas de atividades mostram claramente como operam as estruturas de controle.

Considere o diagrama de atividades de estrutura de sequência da Figura 4.2. Ele contém dois **estados de ações** que representam ações a realizar. Cada estado da ação contém uma **expressão de ação** — por exemplo, “adicionar grade a total” ou “adicionar 1 a counter” — que especifica uma ação particular a realizar. Outras ações poderiam incluir cálculos ou operações de entrada e saída. As setas no diagrama de atividades são chamadas de setas de transição. Essas setas representam **transições**, que indicam a ordem em que ocorrem as ações representadas pelos estados de ação — o programa que implementa as atividades ilustradas pelo diagrama de atividades na Figura 4.2 primeiro adiciona grade a `total`, e, então, adiciona 1 a `counter`.

O **círculo sólido** localizado na parte superior do diagrama de atividades representa o **estado inicial** da atividade — o início do fluxo de trabalho antes de o programa realizar as atividades modeladas. O círculo sólido cercado por um círculo vazio que aparece na parte inferior do diagrama de atividades representa o estado final — o fim do fluxo de trabalho depois que o programa realiza suas atividades.

A Figura 4.2 também inclui retângulos com os cantos superiores direitos dobrados. Essas são chamadas **notas** na UML. As notas são observações explanatórias que descrevem o propósito dos símbolos no diagrama. As notas podem ser utilizadas em qualquer diagrama UML — não só em diagramas de atividades. A Figura 4.2 utiliza notas da UML para mostrar o código C++ associado ao estado de cada ação no diagrama de atividades. Uma **linha pontilhada** conecta cada nota ao elemento que a nota descreve. Os diagramas de atividades normalmente não mostram o código C++ que implementa a atividade. Aqui, utilizamos notas para esse propósito a fim de ilustrar como os diagramas se relacionam ao código C++. Para informações adicionais sobre a UML, veja nosso estudo de caso opcional, que aparece nas seções “Estudo de caso de engenharia de software” no final dos capítulos 1–7, 9, 10, 12 e 13 ou visite www.uml.org.

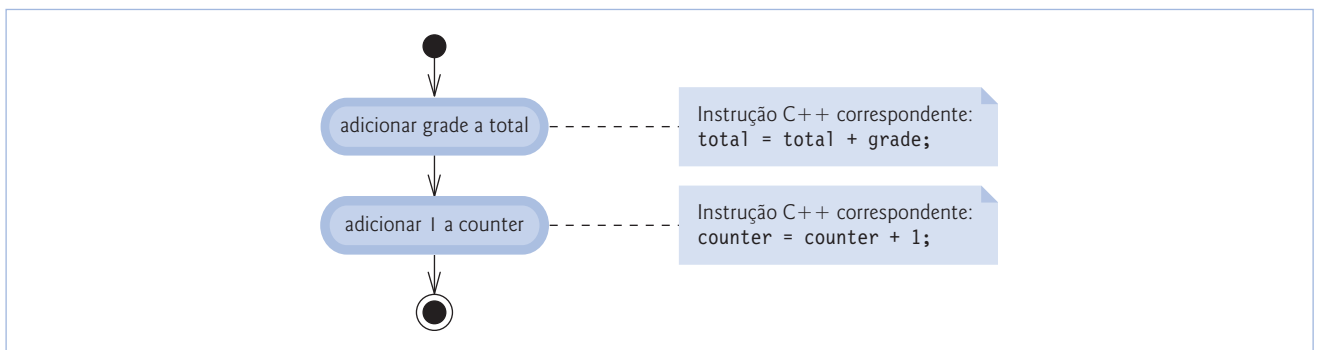


Figura 4.2 Diagrama de atividades da estrutura de sequência.

² Esse documento é mais especificamente conhecido como *NCITS/ISO/IEC 14882-2003 Programming languages — C++* e está disponível para download (por uma taxa) em: webstore.ansi.org/ansidocstore/product.asp?sku=INCITS%2FISO%2FIEC+14882%2D2003.

Instruções de seleção em C++

O C++ fornece três tipos de instruções de seleção (discutidos neste capítulo e no Capítulo 5). A instrução de seleção `if` realiza (seleciona) uma ação se uma condição (predicado) for verdadeira, ou pula a ação se a condição for falsa. A instrução de seleção `if...else` realiza uma ação se uma condição for verdadeira, ou realiza uma ação diferente se a condição for falsa. A instrução de seleção `switch` (Capítulo 5) realiza uma de muitas ações diferentes, dependendo do valor de uma expressão do tipo inteiro.

A instrução de seleção `if` é uma **instrução de uma única seleção** porque seleciona ou ignora uma única ação (ou, como veremos a seguir, um único grupo de ações). A instrução `if...else` é chamada **instrução de seleção dupla** porque seleciona entre duas ações diferentes (ou grupos de ações). A instrução de seleção `switch` é chamada de **instrução de seleção múltipla**, uma vez que seleciona entre muitas ações diferentes (ou grupos de ações).

Instruções de repetição em C++

O C++ fornece três tipos de instruções de repetição (também chamadas de **instruções de loops** ou **loops**) que permitem aos programas realizar instruções repetidamente se uma condição (chamada de **condição de continuação de loop**) permanecer verdadeira. As instruções de repetição são as instruções **`while`**, **`do...while`** e **`for`**. (O Capítulo 5 apresenta as instruções `do...while` e `for`.) As instruções `while` e `for` realizam a ação (ou grupo de ações) no seu corpo zero ou mais vezes — se a condição de continuação de loop for inicialmente falsa, a ação (ou grupo de ações) não será executada. A instrução `do...while` realiza a ação (ou grupo de ações) no seu corpo pelo menos uma vez.

Cada uma das palavras `if`, `else`, `switch`, `while`, `do` e `for` é uma palavra-chave C++. Essas palavras são reservadas pela linguagem de programação C++ para implementar vários recursos, como instruções de controle do C++. As palavras-chave não devem ser utilizadas como identificadores, como nomes de variável. A Figura 4.3 contém uma lista completa de palavras-chave do C++.



Erro comum de programação 4.1

Utilizar uma palavra-chave como um identificador é um erro de sintaxe.

Palavras-chave do C++

Palavras-chave comuns às linguagens de programação C e C++

<code>auto</code>	<code>break</code>	<code>case</code>	<code>char</code>	<code>const</code>
<code>continue</code>	<code>default</code>	<code>do</code>	<code>double</code>	<code>else</code>
<code>enum</code>	<code>extern</code>	<code>float</code>	<code>for</code>	<code>goto</code>
<code>if</code>	<code>int</code>	<code>long</code>	<code>register</code>	<code>return</code>
<code>short</code>	<code>signed</code>	<code>sizeof</code>	<code>static</code>	<code>struct</code>
<code>switch</code>	<code>typedef</code>	<code>union</code>	<code>unsigned</code>	<code>void</code>
<code>volatile</code>	<code>while</code>			

Palavras-chave do C++ somente

<code>and</code>	<code>and_eq</code>	<code>asm</code>	<code>bitand</code>	<code>bitor</code>
<code>bool</code>	<code>catch</code>	<code>class</code>	<code>compl</code>	<code>const_cast</code>
<code>delete</code>	<code>dynamic_cast</code>	<code>explicit</code>	<code>export</code>	<code>false</code>
<code>friend</code>	<code>inline</code>	<code>mutable</code>	<code>namespace</code>	<code>new</code>
<code>not</code>	<code>not_eq</code>	<code>operator</code>	<code>or</code>	<code>or_eq</code>
<code>private</code>	<code>protected</code>	<code>public</code>	<code>reinterpret_cast</code>	<code>static_cast</code>
<code>template</code>	<code>this</code>	<code>throw</code>	<code>true</code>	<code>try</code>
<code>typeid</code>	<code>typename</code>	<code>using</code>	<code>virtual</code>	<code>wchar_t</code>
<code>xor</code>	<code>xor_eq</code>			

Figura 4.3 Palavras-chave do C++.



Erro comum de programação 4.2

Escrever uma palavra-chave com alguma letra maiúscula é um erro de sintaxe. Todas as palavras-chave do C++ têm apenas letras minúsculas.

Resumo de instruções de controle em C++

O C++ contém somente três tipos de estruturas de controle, que daqui para a frente chamaremos instruções de controle: a instrução de sequência, instruções de seleção (três tipos — `if`, `if...else` e `switch`) e instruções de repetição (três tipos — `while`, `for` e `do...while`). Todo programa C++ combina quantas dessas instruções de controle for apropriado para o algoritmo que o programa implementa. Como ocorre com a instrução de sequência da Figura 4.2, você pode modelar cada instrução de controle como um diagrama de atividades. Cada diagrama contém um estado inicial e um estado final, que representam um ponto de entrada e um ponto de saída da instrução de controle, respectivamente. Essas **instruções de controle de entrada única/saída única** facilitam a construção de programas — as instruções de controle são vinculadas conectando-se o ponto de saída de uma instrução ao ponto de entrada da seguinte. Isso é semelhante à maneira como uma criança empilha blocos de construção, portanto chamamos isso de **empilhamento de instruções de controle**. Aprenderemos em breve que há somente uma outra maneira de conectar instruções de controle — chamada de **aninhamento de instrução de controle**, em que uma instrução de controle está contida dentro de outra. Portanto, algoritmos nos programas C++ são construídos a partir de apenas três tipos de instruções de controle, combinadas apenas de duas maneiras. Isso é a essência da simplicidade.



Observação de engenharia de software 4.1

Qualquer programa C++ que criemos pode ser construído a partir de apenas sete tipos de instruções de controle diferentes (sequência, `if`, `if...else`, `switch`, `while`, `do...while` e `for`) combinados apenas de duas maneiras (empilhamento de instruções de controle e aninhamento de instruções de controle).

4.5 Instrução de seleção `if`

Os programas utilizam instruções de seleção para escolher entre cursos alternativos de ações. Por exemplo, suponha que a nota de aprovação de um exame seja 60. A instrução em pseudocódigo

*Se a nota do aluno for maior que ou igual a 60
Imprima 'Passed'*

determina se a condição ‘a nota do aluno for maior que ou igual a 60’ é `true` (verdadeira) ou `false` (falsa). Se a condição for `true`, então ‘Passed’ é impresso e a próxima instrução em pseudocódigo é ‘realizada’ (lembre-se de que pseudocódigo não é uma linguagem de programação real). Se a condição for `false`, a instrução de impressão é ignorada e a próxima instrução em pseudocódigo na sequência é realizada. Observe que a segunda linha dessa instrução de seleção está recuada. Esse recuo é opcional, mas é recomendado uma vez que enfatiza a estrutura inerente de programas estruturados. Quando você converte o pseudocódigo em código C++, o compilador C++ ignora caracteres de espaço em branco (como espaços em branco, tabulações e nova linha) utilizados para recuo e espaçamento vertical.



Boa prática de programação 4.1

Aplicar consistentemente convenções razoáveis de recuo ao longo de todos os seus programas melhora significativamente a legibilidade do programa. Sugerimos três espaços por recuo. Algumas pessoas preferem utilizar tabulações, mas estas podem variar de um editor para outro, fazendo com que um programa escrito em um editor seja alinhado diferentemente quando utilizado com outro.

A instrução *Se* do pseudocódigo precedente pode ser escrita em C++ como

```
if ( grade >= 60 )  
    cout << "Passed";
```

Note que o código C++ apresenta uma íntima correspondência com o pseudocódigo. Essa é uma das propriedades do pseudocódigo que torna essa ferramenta de desenvolvimento de programas tão útil.

A Figura 4.4 ilustra a instrução `if` de uma única seleção. Ele contém o que talvez seja o símbolo mais importante em um diagrama de atividades — o losango, ou **símbolo de decisão**, que indica que uma decisão deve ser tomada. O símbolo de decisão indica que o fluxo de trabalho continuará por um caminho determinado pelas **condições de guarda** do símbolo associado, as quais podem ser verdadeiras ou falsas. Cada seta de transição que sai de um símbolo de decisão tem uma condição de guarda (especificada entre colchetes acima ou ao lado da seta de transição). Se uma condição de guarda for verdadeira, o fluxo de trabalho entra no estado de ação para o qual a seta de transição aponta. Na Figura 4.4, se a nota for maior que ou igual a 60, o programa imprime ‘Passed’ na tela, e, em seguida, passa para o estado final dessa atividade. Se a nota for menor que 60, o programa se dirige imediatamente para o estado final sem exibir uma mensagem.

Aprendemos no Capítulo 1 que as decisões podem ser baseadas em condições contendo operadores relacionais ou de igualdade. De fato, em C++, uma decisão pode ser baseada em qualquer expressão — se a expressão é avaliada como zero, ela é tratada como falsa; se avaliada como não-zero, é tratada como verdadeira. O C++ fornece o tipo de dados **bool** para variáveis que podem armazenar apenas os valores **true** e **false** — cada um deles é uma palavra-chave C++.

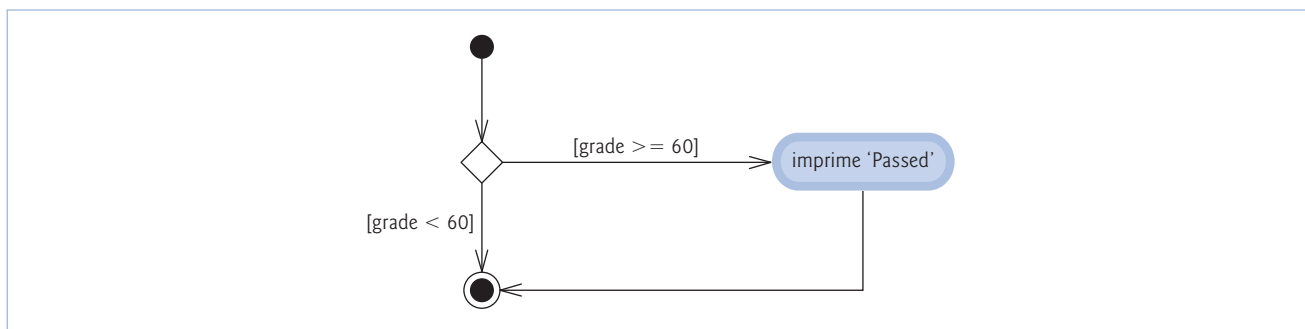


Figura 4.4 Diagrama de atividades de uma instrução de seleção única if.



Dica de portabilidade 4.1

Para compatibilidade com versões anteriores do C, que utilizavam inteiros para valores booleanos, o valor `bool true` também pode ser representado por qualquer valor não-zero (compiladores normalmente utilizam 1) e o valor `bool false` também pode ser representado como o valor zero.

Observe que a instrução `if` é uma instrução de entrada única/saída única. Veremos que os diagramas de atividades para as instruções de controle restantes também contêm estados iniciais, setas de transição, estados de ação que indicam ações a realizar, símbolos de decisão (com condições de guarda associadas) que indicam decisões a serem tomadas e estados finais. Isso é consistente com o **modelo de ação/decisão de programação** que temos enfatizado.

Podemos imaginar sete contêineres, cada um contendo somente diagramas de atividades UML vazios de um dos sete tipos de instruções de controle. A tarefa do programador é, portanto, montar um programa dos diagramas de atividades de cada tipo de instrução de controle que o algoritmo demanda, combinar os diagramas de atividades de apenas duas maneiras possíveis (empilhamento ou aninhamento) e, então, preencher os estados de ação e decisões com expressões de ação e condições de guarda de maneira adequada para formar uma implementação estruturada para o algoritmo. Discutiremos agora a variedade de maneiras em que as ações e decisões podem ser escritas.

4.6 A instrução de seleção dupla if...else

A instrução `if` de uma única seleção realiza uma ação indicada somente quando a condição é `true`; caso contrário, a ação é pulada. A instrução de seleção dupla `if...else` permite que o programador especifique uma ação a realizar quando a condição é `true`, e uma ação diferente a realizar quando a condição é `false`. Por exemplo, a instrução em pseudocódigo

Se a nota do aluno for maior que ou igual a 60

Imprima 'Passed'

Caso contrário

Imprima 'Failed'

imprime 'Passed' se a nota do aluno for maior que ou igual a 60 e imprime 'Failed' se a nota do aluno for menor que 60. Nos dois casos, depois que ocorre a impressão, a próxima instrução em pseudocódigo na sequência é realizada.

A instrução *Se...Caso contrário* do pseudocódigo precedente pode ser escrita em C++ como

```

if ( grade >= 60 )
    cout << "Passed";
else
    cout << "Failed";
  
```

Observe que o corpo do `else` também é recuado. Qualquer que seja a convenção de recuo que você escolher, deve aplicá-la consistentemente por todos os seus programas. É difícil ler programas que não obedecem às convenções de espaçamento uniforme.



Boa prática de programação 4.2

Recue as duas instruções do corpo de uma instrução `if...else`.



Boa prática de programação 4.3

Se existem vários níveis de recuo, cada nível deve ser recuado pela mesma quantidade adicional de espaço.

A Figura 4.5 ilustra o fluxo de controle na instrução `if...else`. Mais uma vez, observe que (além do estado inicial, setas de transição e estado final) os únicos outros símbolos no diagrama de atividades representam estados de ação e decisões. Continuamos a enfatizar

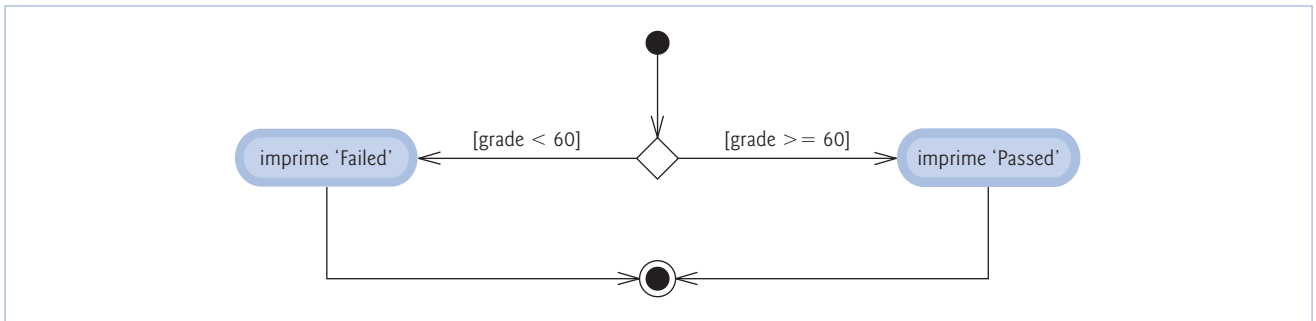


Figura 4.5 Diagrama de atividades de uma instrução de seleção dupla `if...else`.

esse modelo de computação de ação/decisão. Imagine novamente um contêiner profundo de diagramas de atividades UML vazios com instruções de seleção dupla — tantas quantas o programador pode precisar para empilhar e aninhar com os diagramas de atividades de outras instruções de controle para formar uma implementação estruturada de um algoritmo. O programador preenche os estados de ação e símbolos de decisão com expressões de ação e condições de guarda apropriadas ao algoritmo.

Operador condicional (`?:`)

O C++ fornece o **operador condicional** (`?:`), que está intimamente relacionado com a instrução `if...else`. O operador condicional é o único **operador ternário** do C++ — ele aceita três operandos. Os operandos, juntos com o operador condicional, formam uma **expressão condicional**. O primeiro operando é uma condição, o segundo é o valor para a expressão condicional inteira se a condição for `true`, e o terceiro operando é o valor para a expressão condicional inteira se a condição for `false`. Por exemplo, a instrução de saída

```
cout << ( grade >= 60 ? "Passed" : "Failed" );
```

contém uma expressão condicional, `grade >= 60 ? "Passed" : "Failed"`, que é avaliada como a string `"Passed"` se a condição `grade >= 60` for `true`, mas é avaliada como a string `"Failed"` se a condição for `false`. Portanto, a instrução com o operador condicional realiza essencialmente a mesma coisa que a instrução `if...else` anterior. Como veremos, a precedência do operador condicional é baixa, portanto os parênteses na expressão anterior são requeridos.



Dica de prevenção de erro 4.1

Para evitar problemas de precedência (e por clareza), coloque as expressões condicionais (que aparecem em expressões maiores) entre parênteses.

Os valores em uma expressão condicional também podem ser ações a executar. Por exemplo, a seguinte expressão condicional também imprime `"Passed"` ou `"Failed"`:

```
grade >= 60 ? cout << "Passed" : cout << "Failed";
```

A expressão condicional precedente é lida assim: “Se `grade` for maior que ou igual a 60, então `cout << "Passed"`; caso contrário, `cout << "Failed"`.” Essa, também, é comparável à instrução `if...else` precedente. As expressões condicionais podem aparecer em algumas localizações do programa onde as instruções `if...else` não podem.

Instruções `if...else` aninhadas

As **instruções `if...else` aninhadas** testam múltiplos casos colocando instruções de seleção `if...else` dentro de outras instruções de seleção `if...else`. Por exemplo, a seguinte instrução `if...else` em pseudocódigo imprime A para notas de exame maiores que ou igual a 90, B para notas no intervalo 80 a 89, C para notas no intervalo 70 a 79, D para notas no intervalo 60 a 69, e F para todas as outras notas:

Se a nota do aluno for maior que ou igual a 90

Imprima 'A'

Caso contrário

Se a nota do aluno for maior que ou igual a 80

Imprima 'B'

Caso contrário

Se a nota do aluno for maior que ou igual a 70

Imprima 'C'

Caso contrário

Se a nota do aluno for maior que ou igual a 60

Imprima 'D'

Caso contrário

Imprima 'F'

Esse pseudocódigo pode ser escrito em C++ como

```
if ( studentGrade >= 90 ) // 90 e acima obtém "A"
    cout << "A";
else
    if ( studentGrade >= 80 ) // 80-89 obtém "B"
        cout << "B";
    else
        if ( studentGrade >= 70 ) // 70-79 obtém "C"
            cout << "C";
        else
            if ( studentGrade >= 60 ) // 60-69 obtém "D"
                cout << "D";
            else // menor que 60 obtém "F"
                cout << "F";
```

Se studentGrade for maior que ou igual a 90, as primeiras quatro condições serão true, mas somente a instrução cout depois do primeiro teste executará. Depois que cout executa, o programa pula a parte else da instrução if...else 'mais externa'. A maioria dos programadores em C++ prefere escrever a instrução if...else precedente como

```
if ( studentGrade >= 90 ) // 90 e acima obtém "A"
    cout << "A";
else if ( studentGrade >= 80 ) // 80-89 obtém "B"
    cout << "B";
else if ( studentGrade >= 70 ) // 70-79 obtém "C"
    cout << "C";
else if ( studentGrade >= 60 ) // 60-69 obtém "D"
    cout << "D";
else // menor que 60 obtém "F"
    cout << "F";
```

As duas formas são idênticas, exceto quanto ao espaçamento e recuo, que o compilador ignora. A última forma é popular porque evita grande recuo de código para a direita. Esse recuo frequentemente deixa pouco espaço em uma linha, forçando muitas quebras de linhas e diminuindo a legibilidade do programa.



Dica de desempenho 4.1

Uma instrução if...else aninhada pode executar com muito mais rapidez que uma série de instruções if de seleção única por causa da possibilidade de saída prévia depois de uma das condições ser satisfeita.



Dica de desempenho 4.2

Em uma instrução if...else aninhada, teste as condições que têm maior probabilidade de ser true no começo da instrução if...else aninhada. Isso permitirá que a instrução if...else aninhada execute mais rapidamente e saia mais cedo do que ao testar primeiro os casos que ocorrem raramente.

O problema do else oscilante

O compilador C++ sempre associa um else à instrução if imediatamente anterior, a menos que instruído de outro modo pela colocação de chaves ({ e }). Esse comportamento pode levar àquilo que é chamado de **problema do else oscilante**. Por exemplo,

```
if ( x > 5 )
    if ( y > 5 )
        cout << "x and y are > 5";
else
    cout << "x is <= 5";
```

parece indicar que, se x for maior do que 5, a instrução if aninhada determina se y também é maior do que 5. Se então, "x and y are > 5" é saída. Caso contrário, parece que se x não for maior que 5, a parte else do if...else gera a saída de "x is <= 5".

Cuidado! Essa instrução if...else aninhada não é executada como parece. Na verdade, o compilador interpreta a instrução como

```
if ( x > 5 )
    if ( y > 5 )
        cout << "x and y are > 5";
else
    cout << "x is <= 5";
```


em que o corpo do primeiro `if` é um `if...else` aninhado. A instrução `if` externa testa se `x` é maior do que 5. Se for, a execução continuará testando se `y` também é maior que 5. Se a segunda condição for verdadeira, a string adequada — "`x and y are > 5`" — é exibida. Entretanto, se a segunda condição for falsa, a string "`x is <= 5`" é exibida, apesar de sabermos que `x` é maior que 5.

Para forçar a instrução `if...else` aninhada a executar como foi originalmente concebida, devemos escrevê-la como a seguir:

```
if ( x > 5 )
{
    if ( y > 5 )
        cout << "x and y are > 5";
}
else
    cout << "x is <= 5";
```

As chaves (`{}`) indicam ao compilador que a segunda instrução `if` está no corpo da primeira `if` e que o `else` está associado com a primeira `if`. Os exercícios 4.23 e 4.24 investigam o problema do `else` oscilante mais detalhadamente.

Blocos

A instrução de seleção `if` normalmente espera somente uma instrução no seu corpo. De maneira semelhante, as partes `if` e `else` de uma instrução `if...else` esperam apenas uma instrução de corpo. Para incluir várias instruções no corpo de uma parte `if` ou em qualquer parte de uma `if...else`, coloque as instruções entre chaves (`{ e }`). Um conjunto de instruções entre chaves é chamado de **instrução composta** ou **bloco**. Utilizamos o termo 'bloco' deste ponto em diante.



Observação de engenharia de software 4.2

Um bloco pode ser colocado em qualquer lugar em um programa em que uma única instrução pode ser colocada.

O exemplo a seguir inclui um bloco na parte `else` de uma instrução `if...else`:

```
if ( studentGrade >= 60 )
    cout << "Passed.\n";
else
{
    cout << "Failed.\n";
    cout << "You must take this course again.\n";
}
```

Nesse caso, se `studentGrade` é menor que 60, o programa executa ambas as instruções no corpo do `else` e imprime

```
Failed.
You must take this course again.
```

Note as chaves que cercam as duas instruções na cláusula `else`. Essas chaves são importantes. Sem as chaves, a instrução

```
cout << "You must take this course again.\n";
```

ficaria fora do corpo da parte `else` do `if` e executaria independentemente de a nota ser ou não menor que 60. Esse é um exemplo de um erro de lógica.



Erro comum de programação 4.3

Esquecer uma ou ambas as chaves que delimitam um bloco pode levar a erros de sintaxe ou erros de lógica em um programa.



Boa prática de programação 4.4

Colocar sempre as chaves em uma instrução `if... else` (ou qualquer instrução de controle) ajuda a evitar sua omissão acidental, especialmente ao adicionar instruções a uma cláusula `if` ou `else` mais tarde. Para evitar omitir uma ou as duas chaves, alguns programadores preferem digitar as chaves de abertura e fechamento de blocos mesmo antes de digitar as instruções individuais dentro das chaves.

Exatamente como um bloco, uma instrução única pode ser colocada em qualquer lugar; também é possível não ter nenhuma instrução — isso é chamado de **instrução nula** (ou **instrução vazia**). A instrução nula é representada colocando-se um ponto-e-vírgula (`;`) onde normalmente entraria uma instrução.



Erro comum de programação 4.4

Colocar um ponto-e-vírgula depois da condição em uma instrução `if` leva a um erro de lógica em instruções de uma única seleção `if` e a um erro de sintaxe em instruções de seleção dupla `if...else` (quando a parte `if` contém uma instrução de corpo real).

4.7 A instrução de repetição while

Uma **instrução de repetição** (também chamada **instrução de loop** ou simplesmente **loop**) permite ao programador especificar que um programa deve repetir uma ação enquanto alguma condição permanecer verdadeira. A instrução em pseudocódigo

*Enquanto houver mais itens em minha lista de compras
Comprar o próximo item e riscá-lo da minha lista*

descreve a repetição que ocorre durante um passeio de compras. A condição, ‘enquanto houver mais itens em minha lista de compras’ pode ser verdadeira ou falsa. Se ela for verdadeira, então a ação ‘Comprar o próximo item e riscá-lo da minha lista’ é realizada. Essa ação será realizada repetidamente enquanto a condição permanecer verdadeira. A instrução contida na instrução de repetição *Enquanto* constitui o corpo de *Enquanto*, que pode ser uma instrução única ou um bloco. Por fim, a condição se tornaria falsa (quando o último item na lista de compras foi comprado e riscado da lista). Nesse ponto, a repetição termina e a primeira instrução em pseudocódigo depois da instrução de repetição é executada.

Como um exemplo da instrução de repetição *while* e do C++, considere um segmento de programa projetado para localizar a primeira potência de 3 maior que 100. Suponha que a variável do tipo inteiro `product` tenha sido inicializada como 3. Quando a instrução de repetição *while* a seguir terminar de executar, `product` conterá o resultado:

```
int product = 3;

while ( product <= 100 )
    product = 3 * product;
```

Quando a instrução *while* inicia a execução, o valor de `product` é 3. Cada repetição da instrução *while* multiplica o produto por 3, então `product` assume os valores 9, 27, 81 e 243, sucessivamente. Quando `product` torna-se 243, a condição de instrução *while* — `product <= 100` — torna-se *false*. Isso termina a repetição, portanto o valor final de `product` é 243. Nesse ponto, a execução de programa continua com a próxima instrução depois da instrução *while*.



Erro comum de programação 4.5

Não fornecer, no corpo de uma instrução while, uma ação que conseqüentemente faz com que a condição na while se torne falsa normalmente resulta em um erro de lógica chamado loop infinito, no qual a instrução de repetição nunca termina. Isso pode fazer um programa parecer ‘travado’ ou ‘congelado’ se o corpo do loop não contiver instruções que interagem com o usuário.

O diagrama de atividades UML da Figura 4.6 ilustra o fluxo de controle que corresponde à instrução *while* anterior. Mais uma vez, os símbolos no diagrama (além do estado inicial, setas de transição, um estado final e três notas) representam um estado e uma decisão de ação. Esse diagrama também introduz **símbolo de agregação** da UML, que une dois fluxos de atividade a um único. A UML representa o símbolo de agregação e o símbolo de decisão como losangos. Nesse diagrama, o símbolo de agregação une as transições do estado inicial e do estado de ação, assim ambos fluem para a decisão que determina se o loop deve iniciar (ou continuar) a execução. Os símbolos de decisão e agregação podem ser separados pelo número de setas de transição ‘entrantes’ e ‘saindes’. Um símbolo de decisão contém uma seta de transição apontando para o losango e duas ou mais setas de transição apontando a partir do losango para indicar possíveis transições a partir desse ponto. Além disso, cada seta de transição apontando de um símbolo de decisão contém uma condição de guarda ao lado dela. Um símbolo de agregação contém duas ou mais setas de transição apontando para o losango e somente uma seta de transição apontando a partir do losango, para indicar a conexão de múltiplos fluxos de atividades a fim de continuar a atividade. Observe que, diferentemente do símbolo de decisão, o símbolo de agregação não tem uma contraparte no código C++. Nenhuma das setas de transição associadas com um símbolo de agregação contém condições de guarda.

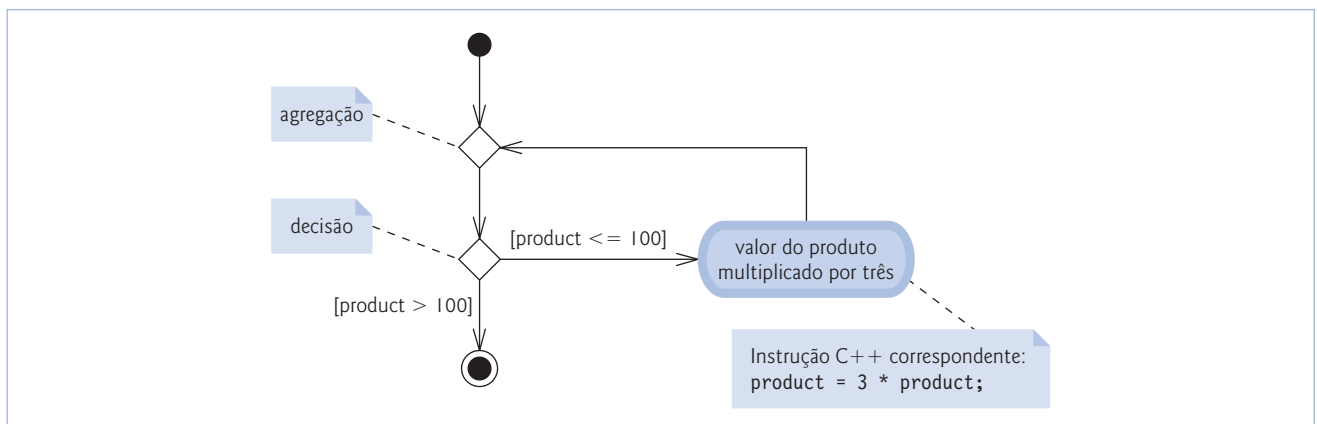


Figura 4.6 Diagrama de atividades UML da instrução de repetição *while*.

O diagrama da Figura 4.6 mostra com clareza a repetição da instrução `while` discutida anteriormente nesta seção. A seta de transição que emerge do estado de ação aponta para o símbolo de agregação, que volta à decisão que é testada a cada passagem pelo loop até a condição de guarda `product > 100` tornar-se verdadeira. Em seguida, a instrução `while` sai (alcança seu estado final) e passa o controle para a próxima instrução na sequência do programa.

Imagine um contêiner profundo de diagramas de atividades de instrução de repetição `while` UML vazios — com a quantidade de diagramas que o programador pode precisar empilhar e aninhar com os diagramas de atividades de outras instruções de controle para formar uma implementação estruturada de um algoritmo. O programador preenche os estados de ação e símbolos de decisão com expressões de ação e condições de guarda apropriadas ao algoritmo.



Dica de desempenho 4.3

Muitas das dicas de desempenho que mencionamos neste texto resultam apenas em pequenas melhoras, portanto o leitor pode ficar tentado a ignorá-las. Entretanto, uma pequena melhora de desempenho para um código que executa muitas vezes em um loop pode resultar em melhora significativa no desempenho geral.

4.8 Formulando algoritmos: repetição controlada por contador

Para ilustrar como os programadores desenvolvem algoritmos, esta seção e a Seção 4.9 resolvem duas variações de um problema que calcula a média da classe. Considere a seguinte declaração do problema:

Uma classe de dez alunos se submeteu a um questionário. As notas (inteiros no intervalo 0 a 100) para esse questionário estão disponíveis. Calcule e exiba o total de todas as notas e a média da classe no questionário.

A média de classe é igual à soma das notas divididas pelo número de alunos. O algoritmo para resolver esse problema em um computador deve inserir cada uma das notas, calcular a média e imprimir o resultado.

Algoritmo em pseudocódigo com repetição controlada por contador

Vamos utilizar o pseudocódigo para listar as ações a executar e especificar a ordem em que essas ações devem ocorrer. Utilizamos **repetição controlada por contador** para inserir as notas uma por vez. Essa técnica utiliza uma variável chamada **contador** para controlar o número de vezes que um grupo de instruções executará (também conhecido como o número de **iterações** do loop).

A repetição controlada por contador é frequentemente chamada **repetição definida**, uma vez que o número de repetições é conhecido antes de o loop começar a executar. Nesse exemplo, a repetição termina quando o contador excede 10. Esta seção apresenta um algoritmo em pseudocódigo completamente desenvolvido (Figura 4.7) e uma versão da classe GradeBook (Figura 4.8–Figura 4.9) que implementa o algoritmo em uma função-membro C++. A seção então apresenta um aplicativo (Figura 4.10) que demonstra o algoritmo em ação. Na Seção 4.9, demonstramos como utilizar o pseudocódigo para desenvolver esse algoritmo a partir do zero.



Observação de engenharia de software 4.3

A experiência tem mostrado que a parte mais difícil de resolver um problema em um computador é desenvolver o algoritmo para a solução. Uma vez que um algoritmo correto foi especificado, o processo de produção de um programa C++ funcional a partir do algoritmo é normalmente simples e direto.

Observe as referências no algoritmo em pseudocódigo da Figura 4.7 a um total e a um contador. Um **total** é uma variável utilizada para acumular a soma de vários valores. Um **contador** é uma variável utilizada para contar — nesse caso, o contador de notas indica qual das dez notas está em vias de ser inserida pelo usuário. Variáveis utilizadas para armazenar totais normalmente são inicializadas

```

1  Configura o total como zero
2  Configura o contador de notas como um
3
4  Enquanto contador de notas for menor ou igual a dez
5      Solicite que o usuário insira a próxima nota
6      Insira a próxima nota
7      Adicione a nota ao total
8      Adicione um ao contador de notas
9
10 Configura a média da classe como o total dividido por dez
11 Imprima o total das notas de todos os alunos da classe
12 Imprima a média da classe

```

Figura 4.7 Algoritmo em pseudocódigo que utiliza repetição controlada por contador para resolver o problema de média de classe.

como zero antes de serem utilizadas em um programa; caso contrário, a soma incluiria o valor anterior armazenado na posição da memória do total.

*Aprimorando a validação de **GradeBook***

Antes de discutirmos a implementação do algoritmo de média da classe, vamos considerar um aprimoramento que fizemos para nossa classe **GradeBook**. Na Figura 3.16, nossa função-membro `setCourseName` validaria o nome do curso primeiro testando se o comprimento do nome do curso era menor que ou igual a 25 caracteres, utilizando uma instrução `if`. Se isso fosse verdadeiro, o nome do curso seria configurado. Esse código era então seguido por outra instrução `if` que testava se o nome do curso tinha um comprimento maior que 25 caracteres (caso em que o nome do curso seria encurtado). Note que a segunda condição da instrução `if` é exatamente o oposto da condição da primeira instrução `if`. Se uma condição é avaliada como `true`, a outra deve ser avaliada como `false`. Tal situação é ideal para uma instrução `if...else`, portanto modificamos nosso código, substituindo as duas instruções `if` por uma instrução `if...else` (linhas 21–28 da Figura 4.9).

*Implementando a repetição controlada por contador na classe **GradeBook***

A classe **GradeBook** (Figura 4.8–Figura 4.9) contém um construtor (declarado na linha 11 da Figura 4.8 e definido nas linhas 12–15 da Figura 4.9) que atribui um valor à variável de instância da classe `courseName` (declarada na linha 17 da Figura 4.8). As linhas 19–29,

```

1 // Figura 4.8: GradeBook.h
2 // Definição da classe GradeBook que determina a média de uma classe.
3 // As funções-membro são definidas no GradeBook.cpp
4 #include <string> // o programa utiliza a classe de string padrão do C++
5 using std::string;
6
7 // definição da classe GradeBook
8 class GradeBook
9 {
10 public:
11     GradeBook( string ); // o construtor inicializa o nome do curso
12     void setCourseName( string ); // função para configurar o nome do curso
13     string getCourseName(); // função para recuperar o nome do curso
14     void displayMessage(); // exibe uma mensagem de boas-vindas
15     void determineClassAverage(); // calcula a média das notas inseridas pelo usuário
16 private:
17     string courseName; // nome do curso para esse GradeBook
18 }; // fim da classe GradeBook

```

Figura 4.8 Problema para calcular a média de uma classe utilizando repetição controlada por contador: arquivo de cabeçalho **GradeBook**.

```

1 // Figura 4.9: GradeBook.cpp
2 // Definições de função-membro para a classe GradeBook que resolve o
3 // programa de média da classe com repetição controlada por contador.
4 #include <iostream>
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include "GradeBook.h" // inclui a definição de classe GradeBook
10
11 // construtor inicializa courseName com String fornecido como argumento
12 GradeBook::GradeBook( string name )
13 {
14     setCourseName( name ); // valida e armazena courseName
15 } // fim do construtor GradeBook

```

Figura 4.9 Problema para calcular a média de uma classe utilizando repetição controlada por contador: arquivo de código-fonte **GradeBook**.

(continua)

```

16
17 // função para configurar o nome do curso;
18 // assegura que o nome do curso tenha no máximo 25 caracteres
19 void GradeBook::setCourseName( string name )
20 {
21     if ( name.length() <= 25 ) // se o nome tiver 25 ou menos caracteres
22         courseName = name; // armazena o nome do curso no objeto
23     else // se o nome tiver mais que 25 caracteres
24     { // configura courseName como os primeiros 25 caracteres do nome de parâmetro
25         courseName = name.substr( 0, 25 ); // seleciona os primeiros 25 caracteres
26         cout << "Name \" " << name << "\" exceeds maximum length (25).\n"
27             << "Limiting courseName to first 25 characters.\n" << endl;
28     } // fim do if...else
29 } // fim da função setCourseName
30
31 // função para recuperar o nome do curso
32 string GradeBook::getCourseName()
33 {
34     return courseName;
35 } // fim da função getCourseName
36
37 // exibe uma mensagem de boas-vindas para o usuário de GradeBook
38 void GradeBook::displayMessage()
39 {
40     cout << "Welcome to the grade book for\n" << getCourseName() << "!\n"
41         << endl;
42 } // fim da função displayMessage
43
44 // determina a média da classe com base em 10 notas inseridas pelo usuário
45 void GradeBook::determineClassAverage()
46 {
47     int total; // soma das notas inseridas pelo usuário
48     int gradeCounter; // número da nota a ser inserida a seguir
49     int grade; // valor da nota inserida pelo usuário
50     int average; // média das notas
51
52     // fase de inicialização
53     total = 0; // inicializa o total
54     gradeCounter = 1; // inicializa o contador de loops
55
56     // fase de processamento
57     while ( gradeCounter <= 10 ) // faz o loop 10 vezes
58     {
59         cout << "Enter grade: "; // solicita entrada
60         cin >> grade; // insere a próxima nota
61         total = total + grade; // adiciona grade a total
62         gradeCounter = gradeCounter + 1; // incrementa o contador por 1
63     } // fim do while
64
65     // fase de término
66     average = total / 10; // divisão de inteiros produz um resultado inteiro
67
68     // exibe o total e a média das notas
69     cout << "\nTotal of all 10 grades is " << total << endl;
70     cout << "Class average is " << average << endl;
71 } // fim da função determineClassAverage

```

Figura 4.9 Problema para calcular a média de uma classe utilizando repetição controlada por contador: arquivo de código-fonte GradeBook.

(continuação)

```

1 // Figura 4.10: fig04_10.cpp
2 // Cria o objeto da classe GradeBook e invoca sua função determineClassAverage.
3 #include "GradeBook.h" // inclui a definição de classe GradeBook
4
5 int main()
6 {
7     // cria o objeto myGradeBook da classe GradeBook e
8     // passa o nome do curso para o construtor
9     GradeBook myGradeBook( "CS101 C++ Programming" );
10
11     myGradeBook.displayMessage(); // exibe a mensagem de boas-vindas
12     myGradeBook.determineClassAverage(); // calcula a média das 10 notas
13     return 0; // indica terminação bem-sucedida
14 } // fim de main

```

```

Welcome to the grade book for
CS101 C++ Programming

```

```

Enter grade: 67
Enter grade: 78
Enter grade: 89
Enter grade: 67
Enter grade: 87
Enter grade: 98
Enter grade: 93
Enter grade: 85
Enter grade: 82
Enter grade: 100

```

```

Total of all 10 grades is 846
Class average is 84

```

Figura 4.10 Problema para calcular a média de uma classe utilizando repetição controlada por contador: criando um objeto da classe GradeBook (Figura 4.8–Figura 4.9) e invocando sua função-membro determineClassAverage.

32–35 e 38–42 da Figura 4.9 definem as funções-membro setCourseName, getCourseName e displayMessage, respectivamente. As linhas 45–71 definem a função-membro determineClassAverage, que implementa o algoritmo de média da classe descrito pelo pseudocódigo na Figura 4.7.

As linhas 47–50 declaram as variáveis locais total, gradeCounter, grade e average como do tipo int. A variável grade armazena a entrada de usuário. Note que as declarações precedentes aparecem no corpo da função-membro determineClassAverage.

Nas versões da classe GradeBook deste capítulo, simplesmente lemos e processamos um conjunto de notas. O cálculo da média é realizado na função-membro determineClassAverage utilizando variáveis locais — não preservamos nenhuma informação sobre as notas dos alunos nas variáveis de instância da classe. No Capítulo 7, “Arrays e vetores”, modificamos a classe GradeBook para manter as notas na memória utilizando uma variável de instância que referencia uma estrutura de dados conhecida como um array. Isso permite que um objeto GradeBook realize vários cálculos sobre o mesmo conjunto de notas sem exigir que o usuário insira as notas múltiplas vezes.



Boa prática de programação 4.5

Separe as declarações de outras instruções nas funções com uma linha em branco para legibilidade.

As linhas 53–54 inicializam total como 0 e gradeCounter como 1. Observe que as variáveis total e gradeCounter são inicializadas antes de ser utilizadas em um cálculo. Normalmente, as variáveis contadoras são inicializadas como zero ou um, dependendo de seu uso (apresentaremos exemplos de cada possibilidade). Uma variável não inicializada contém um **valor ‘lixo’** (também chamado **valor indefinido**) — o último valor armazenado na posição da memória reservada para essa variável. Variáveis grade e average (para a entrada de usuário e a média calculada, respectivamente) não precisam ser inicializadas aqui — seus valores serão atribuídos à medida que forem inseridos ou calculados mais tarde na função.



Erro comum de programação 4.6

Não inicializar contadores e totais pode levar a erros de lógica.



Dica de prevenção de erro 4.2

Inicialize cada contador e total, seja em sua declaração ou em uma instrução de atribuição. Normalmente, os totais são inicializados como 0. Os contadores normalmente são inicializados como 0 ou 1, dependendo de como eles são utilizados (mostraremos exemplos de quando utilizar 0 e quando utilizar 1).



Boa prática de programação 4.6

Declare cada variável em uma linha separada com seu próprio comentário para tornar os programas mais legíveis.

A linha 57 indica que a instrução `while` deve continuar a fazer loop (também chamado **iterar**) contanto que o valor do `gradeCounter` seja menor que ou igual a 10. Enquanto essa condição permanecer verdadeira, a instrução `while` executará repetidamente as instruções entre as chaves que delimitam seu corpo (linhas 58–63).

A linha 59 exibe o prompt "Enter grade: ". Essa linha corresponde à instrução em pseudocódigo 'Solicite que o usuário insira a próxima nota'. A linha 60 lê a nota inserida pelo usuário e a atribui à variável `grade`. Essa linha corresponde à instrução em pseudocódigo 'Insira a próxima nota'. Lembre-se de que a variável `grade` não foi inicializada anteriormente no programa, porque o programa obtém o valor de `grade` a partir do usuário durante cada iteração do loop. A linha 61 adiciona a nova nota inserida pelo usuário ao `total` e atribui o resultado a `total`, que substitui o valor anterior.

A linha 62 adiciona 1 a `gradeCounter` para indicar que o programa processou uma nota e está pronto para inserir a próxima nota fornecida pelo usuário. Incrementando `gradeCounter` por fim faz com que `gradeCounter` exceda 10. Nesse ponto o loop `while` termina porque sua condição (linha 57) torna-se falsa.

Quando o loop termina, a linha 66 realiza o cálculo médio e atribui seu resultado à variável `average`. A linha 69 exibe o texto "Total of all 10 grades is " seguido pelo valor da variável `total`. A linha 70 então exibe o texto "Class average is " seguido pelo valor da variável `average`. A função-membro `determineClassAverage` então retorna o controle à função chamadora (isto é, `main` na Figura 4.10).

Demonstrando a classe **GradeBook**

A Figura 4.10 contém a função `main` desse aplicativo, que cria um objeto da classe `GradeBook` e demonstra suas capacidades. A linha 9 da Figura 4.10 cria um novo objeto `GradeBook` chamado `myGradeBook`. A string na linha 9 é passada para o construtor `GradeBook` (linhas 12–15 da Figura 4.9). A linha 11 da Figura 4.10 chama a função-membro `displayMessage` de `myGradeBook` para exibir uma mensagem de boas-vindas para o usuário. A linha 12 então chama a função-membro `determineClassAverage` de `myGradeBook` para permitir ao usuário inserir dez notas, para as quais a função-membro então calcula e imprime a média — a função-membro realiza o algoritmo mostrado no pseudocódigo da Figura 4.7.

Observações sobre divisão de inteiros e truncamento

O cálculo da média feito pela função-membro `determineClassAverage` em resposta à chamada de função na linha 12 na Figura 4.10 produz um resultado de inteiro. A saída do programa indica que a soma dos valores de nota na execução de exemplo é 846, que, quando dividido por 10, deve ser igual a 84,6 — um número com um ponto de fração decimal. Entretanto, o resultado do cálculo `total / 10` (linha 66 da Figura 4.9) é o inteiro 84, porque `total` e 10 são ambos inteiros. Dividir dois inteiros resulta em divisão de inteiros — qualquer parte fracionária do cálculo é perdida (isto é, **truncada**). Veremos como obter um resultado que inclui um ponto de fração decimal a partir do cálculo da média na próxima seção.



Erro comum de programação 4.7

Assumir que divisão de inteiros arredonda (em vez de truncar) pode levar a resultados incorretos. Por exemplo, $7 \div 4$, que produz 1,75 na aritmética convencional, é truncado para 1 na aritmética de inteiros, em vez de arredondado para 2.

Na Figura 4.9, se a linha 66 utilizasse `gradeCounter` em vez de 10 para o cálculo, a saída desse programa exibiria um valor incorreto, 76. Isso ocorre porque na iteração final da instrução `while`, `gradeCounter` foi incrementado para o valor 11 na linha 62.



Erro comum de programação 4.8

*Utilizar uma variável controlada por contador de um loop em um cálculo depois do loop costuma causar um erro de lógica comum chamado de **erro off-by-one**. Em um loop controlado por contador que é incrementado por um a cada passagem pelo loop, o loop termina quando o valor do contador é um mais alto que seu último valor legítimo (isto é, 11 no caso de contar de 1 a 10).*

4.9 Formulando algoritmos: repetição controlada por sentinela

Vamos generalizar o problema da média da classe. Considere o seguinte problema:

Desenvolva um programa para calcular a média da classe que processe as notas de acordo com um número arbitrário de alunos toda vez que é executado.

No exemplo anterior de cálculo da média da classe, a declaração do problema especificou o número de alunos, assim o número de notas (10) era conhecido antecipadamente. Neste exemplo, nenhuma indicação é dada de quantas notas o usuário irá inserir durante a execução do programa. O programa deve processar um número arbitrário de notas. Como o programa pode determinar quando parar a entrada de notas? Como saber quando calcular e imprimir a média da classe?

Uma maneira de resolver esse problema é utilizar um valor especial chamado **valor de sentinela** (também chamado **valor de sinal**, **valor fictício** ou **valor de flag**) para indicar ‘final de entrada de dados’. O usuário digita notas até a última nota ter sido inserida. O usuário então digita o valor da sentinela para indicar que a última nota foi inserida. A repetição controlada por sentinela é frequentemente chamada **repetição indefinida**, uma vez que o número de repetições não é conhecido antes de o loop iniciar a execução.

Claramente, o valor da sentinela deve ser escolhido de modo que não possa ser confundido com um valor de entrada aceitável. As notas em um questionário normalmente são inteiros não negativos, portanto, para esse problema, -1 é um valor aceitável de sentinela. Portanto, uma execução do programa de média de classe poderia processar um fluxo de entradas como 95, 96, 75, 74, 89 e -1 . O programa então calcularia e imprimiria a média de classe para as notas 95, 96, 75, 74 e 89 (-1 é o valor de sentinela, ele não deve entrar no cálculo da média).



Erro comum de programação 4.9

Escolher um valor de sentinela que também seja um valor legítimo de dados é um erro de lógica.

Desenvolvendo o algoritmo em pseudocódigo com refinamento passo a passo de cima para baixo: a parte superior e o primeiro refinamento

Abordamos o programa de média de classe com uma técnica chamada **refinamento passo a passo de cima para baixo**, uma técnica que é essencial para o desenvolvimento de programas bem estruturados. Iniciamos com uma representação em pseudocódigo do **topo** — uma única instrução que fornece a função geral do programa:

Determine a média de classe para o questionário

O topo é, em efeito, uma representação *completa* de um programa. Infelizmente, o topo (como nesse caso) raramente transmite detalhes suficientes a partir dos quais escrever um programa. Então, agora iniciamos o processo de refinamento. Dividimos o topo em uma série de tarefas menores e as listamos na ordem em que elas precisam ser realizadas. Isso resulta no **primeiro refinamento** que se segue.

Inicialize as variáveis

Insira, some e conte as notas do exame

Calcule e imprima o total de todas as notas de aluno e a média da classe

Esse refinamento utiliza somente a estrutura de sequência — os passos listados devem ser executados na ordem, um depois do outro.



Observação de engenharia de software 4.4

Cada refinamento, bem como o topo, é uma especificação completa do algoritmo; somente o nível de detalhe varia.



Observação de engenharia de software 4.5

Muitos programas podem ser divididos logicamente em três fases: uma fase de inicialização que inicializa as variáveis do programa; uma fase de processamento que insere os valores dos dados e ajusta as variáveis do programa (como contadores e totais) de maneira correspondente; e uma fase de término que calcula e gera a saída dos resultados finais.

Prosseguindo para o segundo refinamento

A “Observação de engenharia de software” anterior costuma ser tudo o que você precisa para o primeiro refinamento no processo de cima para baixo. A fim de prosseguir para o próximo nível de refinamento, isto é, o **segundo refinamento**, definimos variáveis específicas. Neste exemplo, precisamos de um total dos números, uma contagem de quantos números foram processados, uma variável para receber o valor de cada nota à medida que é inserida pelo usuário e uma variável para armazenar a média calculada. A instrução em pseudocódigo

Inicialize as variáveis

pode ser refinada desta maneira:

Inicialize total como zero

Inicialize o contador como zero

Somente as variáveis *total* e *contador* precisam ser inicializadas antes de serem utilizadas. As variáveis *average* e *grade* (para a média calculada e a entrada do usuário, respectivamente) não precisam ser inicializadas, uma vez que seus valores serão substituídos à medida que são calculados ou inseridos.

A instrução em pseudocódigo

Insira, some e conte as notas do exame

requer uma instrução de repetição (isto é, um loop) que insere sucessivamente cada nota. Não conhecemos antecipadamente quantas notas devem ser processadas, assim utilizaremos a repetição controlada por sentinela. O usuário insere as notas legítimas uma por vez. Depois de inserir a última nota legítima, o usuário insere o valor de sentinela. O programa faz um teste para o valor de sentinela depois de cada nota ser inserida e termina o loop quando o usuário insere o valor de sentinela. O segundo refinamento da instrução em pseudocódigo precedente é então

Solicite que o usuário insira a primeira nota

Insira a primeira nota (possivelmente o sentinela)

Enquanto o usuário não inserir o sentinela

Adicione essa nota à soma total

Adicione um ao contador de notas

Solicite que o usuário insira a próxima nota

Insira a próxima nota (possivelmente a sentinela)

No pseudocódigo, não utilizamos chaves em torno das instruções que formam o corpo da estrutura *Enquanto*. Nós simplesmente re-cuamos as instruções abaixo da *Enquanto* para mostrar que pertencem ao *Enquanto*. Novamente, o pseudocódigo é apenas um auxílio informal ao desenvolvimento de programa.

A instrução em pseudocódigo

Calcule e imprima o total de todas as notas de aluno e a média da classe

pode ser refinada desta maneira:

Se o contador não for igual a zero

Configure a média como o total dividido pelo contador

Imprima o total das notas de todos os alunos da classe

Imprima a média da classe

caso contrário

Imprima 'Nenhuma nota foi inserida'

Somos cuidadosos aqui para testar a possibilidade de divisão por zero — normalmente, um **erro de lógica fatal** que, se não detectado, faria com que o programa falhasse (freqüentemente chamado de **'bombing'** ou **'crashing'**). O segundo refinamento completo do pseudocódigo para o problema da média da classe é mostrado na Figura 4.11.

```

1  Inicialize total como zero
2  Inicialize contador como zero
3
4  Solicite que o usuário insira a primeira nota
5  Insira a primeira nota (possivelmente o sentinela)
6
7  Enquanto o usuário não inserir o sentinela
8      Adicione essa nota à soma total
9      Adicione um ao contador de notas
10     Solicite que o usuário insira a próxima nota
11     Insira a próxima nota (possivelmente a sentinela)
12
13  Se o contador não for igual a zero
14      Configure a média como o total dividido pelo contador
15      Imprima o total das notas de todos os alunos da classe
16      Imprima a média da classe
17  caso contrário
18      Imprima 'Nenhuma nota foi inserida'
```

Figura 4.11 Algoritmo em pseudocódigo do problema de média da classe com repetição controlada por sentinela.



Erro comum de programação 4.10

Uma tentativa de dividir por zero normalmente causa um erro fatal de tempo de execução.



Dica de prevenção de erro 4.3

Ao realizar divisão por uma expressão cujo valor poderia ser zero, teste explicitamente para essa possibilidade e trate-a apropriadamente em seu programa (como imprimindo uma mensagem de erro) em vez de permitir que ocorra um erro fatal.

Nas figuras 4.7 e 4.11, incluímos algumas linhas completamente em branco e recuos no pseudocódigo para torná-lo mais legível. As linhas em branco separam os algoritmos em pseudocódigo em suas várias fases e o recuo enfatiza os corpos de instrução de controle.

O algoritmo em pseudocódigo na Figura 4.11 resolve o problema da média de classe mais geral. Esse algoritmo foi desenvolvido após apenas dois níveis de refinamento. Às vezes são necessários mais níveis.



Observação de engenharia de software 4.6

Termine o processo de refinamento passo a passo de cima para baixo quando o algoritmo em pseudocódigo for especificado em detalhes suficientes para você ser capaz de converter o pseudocódigo em C++. Normalmente, implementar o programa C++ é então simples e direto.



Observação de engenharia de software 4.7

Muitos programadores experientes escrevem programas sem jamais utilizar ferramentas de desenvolvimento de programa como pseudocódigo. Esses programadores acreditam que seu objetivo final é resolver o problema em um computador e que escrever pseudocódigo só retarda a produção de saídas finais. Embora esse método possa funcionar para problemas simples e familiares, ele pode levar a sérias dificuldades em projetos complexos e grandes.

Implementando a repetição controlada por sentinela na classe **GradeBook**

As figuras 4.12 e 4.13 mostram a classe **GradeBook** do C++ contendo a função-membro `determineClassAverage` que implementa o algoritmo em pseudocódigo da Figura 4.11 (essa classe é demonstrada na Figura 4.14). Embora cada nota inserida seja um inteiro, o cálculo da média provavelmente produz um número com um ponto de fração decimal — em outras palavras, um número real ou **número de ponto flutuante** (por exemplo, 7,33, 0,0975 ou 1000,12345). O tipo `int` não pode representar tal número, portanto essa classe deve utilizar outro tipo para fazer isso. O C++ fornece vários tipos de dados para armazenar números de ponto flutuante na memória, incluindo **float** e **double**. A principal diferença entre esses tipos é que, comparadas com variáveis `float`, as variáveis `double` podem armazenar números com maior magnitude e mais detalhes (isto é, mais dígitos à direita do ponto de fração decimal — também conhecido como **precisão** do número). Esse programa introduz um operador especial chamado **operador de coerção** para forçar o cálculo da média a produzir um resultado numérico de ponto flutuante. Esses recursos serão explicados em detalhes quando discutirmos o programa.

```

1 // Figura 4.12: GradeBook.h
2 // Definição da classe GradeBook que determina a média de uma classe.
3 // As funções-membro são definidas no GradeBook.cpp
4 #include <string> // o programa utiliza classe de string padrão C++
5 using std::string;
6
7 // definição da classe GradeBook
8 class GradeBook
9 {
10 public:
11     GradeBook( string ); // o construtor inicializa o nome do curso
12     void setCourseName( string ); // função para configurar o nome do curso
13     string getCourseName(); // função para recuperar o nome do curso
14     void displayMessage(); // exibe uma mensagem de boas-vindas
15     void determineClassAverage(); // calcula a média das notas inseridas pelo usuário
16 private:
17     string courseName; // nome do curso para esse GradeBook
18 }; // fim da classe GradeBook

```

Figura 4.12 Problema para calcular média da classe utilizando repetição controlada por sentinela: arquivo de cabeçalho **GradeBook**.

Neste exemplo, vemos que as instruções de controle podem ser empilhadas umas sobre as outras (na sequência) assim como uma criança empilha blocos de construção. A instrução `while` (linhas 67–75 da Figura 4.13) é imediatamente seguida por uma instrução `if...else` (linhas 78–90) na sequência. Boa parte do código nesse programa é idêntica ao código na Figura 4.9, portanto nos concentramos nos novos recursos e questões.

```

1 // Figura 4.13: GradeBook.cpp
2 // Definições de função-membro para a classe GradeBook que resolve o
3 // programa de média de classe com repetição controlada por sentinela.
4 #include <iostream>
5 using std::cout;
6 using std::cin;
7 using std::endl;
8 using std::fixed; // assegura que o ponto de fração decimal seja exibido
9
10 #include <iomanip> // manipuladores de fluxo parametrizados
11 using std::setprecision; // configura a precisão da saída numérica
12
13 // inclui a definição da classe GradeBook de GradeBook.h
14 #include "GradeBook.h"
15
16 // construtor inicializa courseName com string fornecido como argumento
17 GradeBook::GradeBook( string name )
18 {
19     setCourseName( name ); // valida e armazena courseName
20 } // fim do construtor GradeBook
21
22 // função para configurar o nome do curso;
23 // assegura que o nome do curso tenha no máximo 25 caracteres
24 void GradeBook::setCourseName( string name )
25 {
26     if ( name.length() <= 25 ) // se o nome tiver 25 ou menos caracteres
27         courseName = name; // armazena o nome do curso no objeto
28     else // se o nome tiver mais que 25 caracteres
29     { // configura courseName como os primeiros 25 caracteres do nome de parâmetro
30         courseName = name.substr( 0, 25 ); // seleciona os primeiros 25 caracteres
31         cout << "Name \"" << name << "\" exceeds maximum length (25).\n"
32             << "Limiting courseName to first 25 characters.\n" << endl;
33     } // fim do if...else
34 } // fim da função setCourseName
35
36 // função para recuperar o nome do curso
37 string GradeBook::getCourseName()
38 {
39     return courseName;
40 } // fim da função getCourseName
41
42 // exibe uma mensagem de boas-vindas para o usuário de GradeBook
43 void GradeBook::displayMessage()
44 {
45     cout << "Welcome to the grade book for\n" << getCourseName() << "!\n"
46         << endl;
47 } // fim da função displayMessage
48
49 // determina a média da classe com base em 10 notas inseridas pelo usuário
50 void GradeBook::determineClassAverage()

```

Figura 4.13 Problema para calcular média da classe utilizando repetição controlada por sentinela: arquivo de código-fonte GradeBook.

(continua)

```

51 {
52     int total; // soma das notas inseridas pelo usuário
53     int gradeCounter; // número de notas inseridas
54     int grade; // valor da nota
55     double average; // número com ponto de fração decimal para a média
56
57     // fase de inicialização
58     total = 0; // inicializa o total
59     gradeCounter = 0; // inicializa o contador de loops
60
61     // fase de processamento
62     // solicita entrada e lê a nota do usuário
63     cout << "Enter grade or -1 to quit: ";
64     cin >> grade; // insere nota ou valor de sentinela
65
66     // faz um loop até ler o valor de sentinela inserido pelo usuário
67     while ( grade != -1 ) // enquanto a nota não é -1
68     {
69         total = total + grade; // adiciona grade a total
70         gradeCounter = gradeCounter + 1; // incrementa contador
71
72         // solicita entrada e lê a próxima nota fornecida pelo usuário
73         cout << "Enter grade or -1 to quit: ";
74         cin >> grade; // insere nota ou valor de sentinela
75     } // fim do while
76
77     // fase de término
78     if ( gradeCounter != 0 ) // se usuário inseriu pelo menos uma nota...
79     {
80         // calcula a média de todas as notas inseridas
81         average = static_cast< double >( total ) / gradeCounter;
82
83         // exibe o total e a média (com dois dígitos de precisão)
84         cout << "\nTotal of all " << gradeCounter << " grades entered is "
85             << total << endl;
86         cout << "Class average is " << setprecision( 2 ) << fixed << average
87             << endl;
88     } // fim do if
89     else // nenhuma nota foi inserida, assim gera a saída da mensagem apropriada
90         cout << "No grades were entered" << endl;
91 } // fim da função determineClassAverage

```

Figura 4.13 Problema para calcular média da classe utilizando repetição controlada por sentinela: arquivo de código-fonte GradeBook.

(continuação)

A linha 55 declara a variável `double` `average`. Lembre-se de que utilizamos uma variável `int` no exemplo precedente para armazenar a média da classe. Utilizar o tipo `double` no exemplo atual permite armazenar o resultado do cálculo da média da classe como um número de ponto flutuante. A linha 59 inicializa a variável `gradeCounter` como 0, porque nenhuma nota foi ainda inserida. Lembre-se de que esse programa utiliza repetição controlada por sentinela. Para manter um registro exato do número das notas inseridas, o programa incrementa a variável `gradeCounter` somente quando o usuário inserir um valor de nota válido (isto é, não o valor de sentinela) e o programa completa o processamento da nota. Por fim, note que ambas as instruções de entrada (linha 64 e 74) são precedidas por uma instrução de saída que solicita a entrada ao usuário.



Boa prática de programação 4.7

Solicite cada entrada de teclado ao usuário. O prompt deve indicar a forma da entrada e qualquer valor especial de entrada. Por exemplo, em um loop controlado por sentinela, os prompts solicitando entrada de dados devem lembrar explicitamente ao usuário qual é o valor da sentinela.


```

1 // Figura 4.14: fig04_14.cpp
2 // Cria o objeto da classe GradeBook e invoca sua função-membro determineClassAverage
3
4 // inclui a definição da classe GradeBook de GradeBook.h
5 #include "GradeBook.h"
6
7 int main()
8 {
9     // cria o objeto myGradeBook da classe GradeBook e
10    // passa o nome do cursor para o construtor
11    GradeBook myGradeBook( "CS101 C++ Programming" );
12
13    myGradeBook.displayMessage(); // exibe a mensagem de boas-vindas
14    myGradeBook.determineClassAverage(); // calcula a média das 10 notas
15    return 0; // indica terminação bem-sucedida
16 } // fim de main

```

Welcome to the grade book for
CS101 C++ Programming

Enter grade or -1 to quit: 97
Enter grade or -1 to quit: 88
Enter grade or -1 to quit: 72
Enter grade or -1 to quit: -1

Total of all 3 grades entered is 257
Class average is 85.67

Figura 4.14 Problema para calcular média da classe utilizando repetição controlada por sentinela: criando um objeto da classe GradeBook (Figura 4.12–Figura 4.13) e invocando sua função-membro `determineClassAverage`.

Lógica do programa para repetição controlada por sentinela versus repetição controlada por contador

Compare a lógica do programa para a repetição controlada por sentinela nesse aplicativo com a da repetição controlada por contador na Figura 4.9. Na repetição controlada por contador, cada iteração da instrução `while` (linhas 57–63 da Figura 4.9) lê um valor fornecido pelo usuário de acordo com o número especificado de iterações. Na repetição controlada por sentinela, o programa lê o primeiro valor (linhas 63–64 da Figura 4.13) antes de alcançar o `while`. Esse valor determina se o fluxo do programa de controle deve entrar no corpo do `while`. Se a condição do `while` for falsa, o usuário inseriu o valor de sentinela, portanto o corpo do `while` não é executado (isto é, nenhuma nota foi inserida). Se, por outro lado, a condição for verdadeira, o corpo inicia a execução e o loop adiciona o valor de `grade` a `total` (linha 69). As linhas 73–74 no corpo do loop inserem então o próximo valor fornecido pelo usuário. Em seguida, o controle do programa alcança a chave direita de fechamento `}` do corpo na linha 75, assim a execução continua com o teste da condição do `while` (linha 67). A condição utiliza a entrada de `grade` mais recentemente inserida pelo usuário para determinar se o corpo do loop deve executar novamente. Observe que o valor da variável `grade` é sempre a entrada do usuário imediatamente antes de o programa testar a condição `while`. Isso permite que o programa determine se o valor recém-inserido é o valor de sentinela *antes* de o programa processar esse valor (isto é, adiciona-o a `total` e incrementa `gradeCounter`). Se o valor de sentinela for inserido, o loop termina e o programa não adiciona `-1` a `total`.

Depois que o loop termina, a instrução `if...else` nas linhas 78–90 é executada. A condição na linha 78 determina se quaisquer notas foram inseridas. Se nenhuma nota foi inserida, a parte `else` (linhas 89–90) da instrução `if...else` executa e exibe a mensagem "No grades were entered" e a função-membro retorna o controle à função chamadora.

Note o bloco no loop `while` da Figura 4.13. Sem as chaves, as últimas três instruções no corpo do loop cairiam fora do loop, fazendo o computador interpretar esse código incorretamente, como segue:

```

// faz um loop até ler o valor de sentinela inserido pelo usuário
while ( grade != -1 )
    total = total + grade; // adiciona grade a total
    gradeCounter = gradeCounter + 1; // incrementa counter

```

```
// solicita entrada e lê a próxima nota fornecida pelo usuário
cout << "Enter grade or -1 to quit: ";
cin >> grade;
```

Isso causaria um loop infinito no programa se o usuário não inserisse -1 para a primeira nota (na linha 64).



Erro comum de programação 4.11

Omitir as chaves que delimitam um bloco pode levar a erros de lógica, como loops infinitos. Para evitar esse problema, alguns programadores incluem o corpo de cada instrução de controle dentro de chaves, mesmo que o corpo contenha somente uma única instrução.

Precisão de número de ponto flutuante e requisitos de memória

As variáveis de tipo `float` representam **números de ponto flutuante de precisão simples** e têm sete dígitos significativos na maioria dos sistemas de 32 bits de hoje. As variáveis de tipo `double` representam **números de ponto flutuante de dupla precisão**. Essas requerem duas vezes mais memória que as variáveis `float` e hoje fornecem 15 dígitos significativos na maioria dos sistemas de 32 bits — aproximadamente o dobro da precisão das variáveis `float`. Para o intervalo de valores requerido pela maioria dos programas, as variáveis de tipo `float` devem bastar, mas você pode utilizar `double` para ‘trabalhar com segurança’. Em alguns programas, até mesmo as variáveis de tipo `double` serão inadequadas — esses programas estão além do escopo deste livro. A maioria dos programadores representa números de ponto flutuante com o tipo `double`. De fato, o C++ trata todos os números de ponto flutuante que você digita no código-fonte de um programa (como 7,33 e 0,0975) como valores `double` por padrão. Esses valores no código-fonte são conhecidos como **constantes de ponto flutuante**. Consulte o Apêndice C, “Tipos fundamentais”, para os intervalos de valores de `floats` e `doubles`.

Os números de ponto flutuante costumam surgir como resultado de divisão. Na aritmética convencional, quando dividimos 10 por 3, o resultado é 3,333333..., com a seqüência de 3s repetindo-se infinitamente. O computador aloca apenas uma quantidade fixa de espaço para armazenar tal valor, portanto evidentemente o valor de ponto flutuante armazenado somente pode ser uma aproximação.



Erro comum de programação 4.12

Utilizar números de ponto flutuante de uma maneira que supõe que eles são representados exatamente (por exemplo, utilizando-os em comparações de igualdade) pode levar a resultados incorretos. Os números de ponto flutuante são representados apenas aproximadamente pela maioria dos computadores.

Embora os números de ponto flutuante não sejam sempre 100% precisos, eles têm numerosas aplicações. Por exemplo, quando falamos de uma temperatura ‘normal’ de corpo de 98,6°F, não precisamos ser tão precisos a ponto de envolver um grande número de dígitos. Quando lemos a temperatura de 98,6°F em um termômetro, ela pode, de fato, ser 98,5999473210643°F. Chamar simplesmente esse número de 98,6°F serve para a maioria dos aplicativos que medem temperaturas de corpo. Devido à natureza imprecisa dos números de ponto flutuante, o tipo `double` é preferido ao tipo `float` porque as variáveis `double` podem representar números de ponto flutuante com mais exatidão. Por essa razão, utilizamos o tipo `double` por todo o livro.

Convertendo entre tipos fundamentais explícita e implicitamente

A variável `average` é declarada como tipo `double` (linha 55 da Figura 4.13) para capturar o resultado fracionário de nosso cálculo. Entretanto, `total` e `gradeCounter` são ambas variáveis do tipo inteiro. Lembre-se de que dividir dois inteiros resulta em divisão de inteiro, em que qualquer parte fracionária do cálculo é perdida (isto é, **truncada**). Na seguinte instrução:

```
average = total / gradeCounter;
```

o cálculo de divisão é realizado primeiro, então a parte fracionária do resultado é perdida antes de ela ser atribuída a `average`. Para realizar um cálculo de ponto flutuante com valores de inteiro, devemos criar valores temporários que são números de ponto flutuante para o cálculo. O C++ fornece o **operador de coerção unário** para realizar essa tarefa. A linha 81 utiliza o operador de coerção `static_cast< double >( total )` para criar uma cópia de ponto flutuante *temporária* de seu operando entre parênteses — `total`. A utilização de um operador de coerção dessa maneira é chamada de **conversão explícita**. O valor armazenado em `total` ainda é um inteiro.

O cálculo agora consiste em um valor de ponto flutuante (a versão `double` temporária de `total`) dividida pelo inteiro `gradeCounter`. O compilador C++ sabe avaliar somente expressões em que os tipos de dados dos operandos são idênticos. Para assegurar que os operandos sejam do mesmo *tipo*, o compilador realiza uma operação chamada **promoção** (também chamada **conversão implícita**) em operandos selecionados. Por exemplo, em uma expressão contendo valores dos tipos de dados `int` e `double`, o C++ **promove** os operandos `int` a valores `double`. No nosso exemplo, estamos tratando `total` como um `double` (utilizando o operador de coerção unário), então o compilador promove `gradeCounter` a `double`, permitindo que o cálculo seja realizado — o resultado da divisão de ponto flutuante é atribuído a `average`. No Capítulo 6, “Funções e uma introdução à recursão”, discutimos todos os tipos de dados fundamentais e sua ordem de promoção.



Erro comum de programação 4.13

O operador de coerção pode ser utilizado para converter entre tipos numéricos fundamentais, como `int` e `double`, e entre tipos de classe relacionados (como discutiremos no Capítulo 13, “Programação orientada a objetos. Polimorfismo”). Aplicar uma coerção ao tipo errado pode causar erros de compilação ou erros de tempo de execução.

Os operadores de coerção estão disponíveis para o uso com todos os tipos de dados e também com os tipos de classe. O operador `static_cast` é formado pela palavra-chave seguinte `static_cast` com colchetes angulares (`<` e `>`) em torno de um nome de tipo de dados. O operador de coerção é um **operador unário** — um operador que aceita apenas um operando. No Capítulo 2, estudamos os operadores aritméticos binários. O C++ também suporta versões unárias dos operadores mais (+) e menos (-), para que o programador possa escrever expressões como `-7` ou `+5`. Os operadores de coerção têm precedência mais alta que outros operadores unários, como + unário e o - unário. Essa precedência é mais alta do que a dos **operadores multiplicativos** *, / e %, e mais baixa que a de parênteses. Indicamos o operador de coerção com a notação `static_cast<tipo>()` em nossos gráficos de precedência (ver, por exemplo, Figura 4.22).

Formatação de números de ponto flutuante

As capacidades de formatação na Figura 4.13 são brevemente discutidas aqui e explicadas em profundidade no Capítulo 15, “Entrada/saída de fluxo”. A chamada a **setprecision** na linha 86 (com um argumento de 2) indica que variável `double average` deve ser impressa com dois dígitos de **precisão** à direita do ponto de fração decimal (por exemplo, 92,37). Essa chamada é referida como um **manipulador de fluxo parametrizado** (por causa do 2 entre parênteses). Os programas que utilizam essas chamadas devem conter a diretiva de pré-processador (linha 10)

```
#include <iomanip>
```

A linha 11 especifica os nomes do arquivo de cabeçalho `<iomanip>` que são utilizados nesse programa. Observe que `endl` é um **manipulador de fluxo não parametrizado** (porque não é seguido por um valor ou expressão entre parênteses) e não requer o arquivo de cabeçalho `<iomanip>`. Se a precisão não é especificada, a saída dos valores de ponto flutuante tem normalmente seis dígitos de precisão (isto é, a **precisão-padrão** na maioria dos sistemas de 32 bits hoje), embora vejamos uma exceção para isso um pouco mais à frente.

O manipulador de fluxo **fixed** (linha 86) indica que os valores de ponto flutuante devem ser enviados para a saída no chamado **formato de ponto fixo**, em oposição à **notação científica**. A notação científica é uma maneira de exibir um número como um número de ponto flutuante entre os valores de 1 e 10, multiplicado por uma potência de 10. Por exemplo, o valor 3.100 seria exibido em notação científica como $3,1 \times 10^3$. A notação científica é útil para exibir valores muito grandes ou muito pequenos. A formatação utilizando a notação científica é discutida ainda mais no Capítulo 15. A formatação de ponto fixo, por outro lado, é utilizada para forçar um número de ponto flutuante a exibir um número específico de dígitos. Especificar a formatação de ponto fixo também força a impressão de ponto de fração decimal e zeros finais, mesmo que o valor seja um número inteiro, como 88,00. Sem opção de formatação de ponto fixo, esse valor é impresso em C++ como 88 sem os zeros finais e sem o ponto de fração decimal. Quando os manipuladores de fluxo `fixed` e `setprecision` são utilizados em um programa, o valor impresso é **arredondado** para o número de posições decimais indicado pelo valor passado para `setprecision` (por exemplo, o valor 2 na linha 86), embora o valor na memória permaneça inalterado. Por exemplo, as saídas dos valores 87,946 e 67,543 são como 87,95 e 67,54, respectivamente. Observe que também é possível forçar um ponto de fração decimal a aparecer utilizando manipulador de fluxo **showpoint**. Se `showpoint` é especificado sem `fixed`, zeros finais não serão impressos. Como `endl`, os manipuladores de fluxo `fixed` e `showpoint` não são parametrizados e não requerem o arquivo de cabeçalho `<iomanip>`. Ambos podem ser encontrados no cabeçalho `<iostream>`.

As linhas 86 e 87 da Figura 4.13 geram a saída da média da classe. Nesse exemplo, exibimos a média de classe arredondada para centésimo mais próximo e realizamos a saída com exatamente dois dígitos à direita do ponto de fração decimal. O manipulador de fluxo parametrizado (linha 86) indica que o valor da variável `average` deve ser exibido com dois dígitos de precisão à direita do ponto de fração decimal — indicado por `setprecision( 2 )`. As três notas entraram durante a execução de exemplo do programa na Figura 4.14 totais 257, que produz as médias 85,666666... O manipulador de fluxo parametrizado `setprecision` faz com que o valor seja arredondado para o número especificado de dígitos. Nesse programa, a média é arredondada para a posição de centésimos e exibida como 85.67.

4.10 Formulando algoritmos: instruções de controle aninhadas

Para o próximo exemplo, mais uma vez formulamos um algoritmo utilizando o pseudocódigo e o refinamento passo a passo de cima para baixo e escrevemos um programa C++ correspondente. Vimos que as instruções de controle podem ser empilhadas umas sobre as outras (em sequência) assim como uma criança empilha blocos de construção. Nesse estudo de caso, examinaremos a outra maneira estruturada de instruções de controle poderem ser conectadas, a saber, **aninhando** uma instrução de controle dentro de outra.

Considere a seguinte declaração do problema:

Uma faculdade oferece um curso que prepara os candidatos a obter licença estadual para corretores de imóveis. No ano passado, dez alunos que concluíram esse curso prestaram o exame. A universidade quer saber como foi o desempenho dos seus alunos nesse exame. Você foi contratado para escrever um programa que resuma os resultados. Para tanto, você recebeu uma lista com dez desses alunos. Ao lado de cada nome é escrito 1 se o aluno passou no exame ou 2 se o aluno foi reprovado.

Seu programa deve analisar os resultados do exame assim:

1. *Insira cada resultado de teste (isto é, 1 ou 2). Exiba a mensagem de solicitação 'Inserir resultado' toda vez que o programa solicitar outro resultado de teste.*
2. *Conte o número de cada tipo de resultado.*
3. *Exiba um resumo dos resultados do teste indicando o número de alunos aprovados e reprovados.*
4. *Se mais de oito alunos foram aprovados no exame, imprima a mensagem 'Aumentar a mensalidade escolar'.*

Depois de ler a declaração do problema cuidadosamente, fazemos estas observações:

1. O programa deve processar resultados de teste para dez alunos. Um loop controlado por contador pode ser utilizado porque o número de resultados do teste é conhecido antecipadamente.
2. Cada resultado do teste é um número — 1 ou 2. Toda vez que o programa ler um resultado, deve determinar se o número é 1 ou 2. Em nosso algoritmo, testamos se o número é 1. Se o número não for um 1, supomos que ele seja um 2. (O Exercício 4.20 considera as consequências dessa suposição.)
3. Dois contadores são utilizados para monitorar os resultados do exame — um para contar o número de alunos que foram aprovados no exame e outro para contar o número de alunos que foram reprovados no exame.
4. Depois que o programa processou todos os resultados, ele deve decidir se mais de oito alunos foram aprovados no exame.

Vamos prosseguir com o refinamento passo a passo de cima para baixo. Iniciamos com uma representação do pseudocódigo da parte superior:

Analise os resultados do exame e decida se a mensalidade escolar deve ser elevada

Mais uma vez, é importante enfatizar que a parte superior é uma representação *completa* do programa, mas vários refinamentos são provavelmente necessários antes que o pseudocódigo possa transformar-se naturalmente em um programa C++.

Nosso primeiro refinamento é

Inicialize as variáveis

Insira os 10 resultados dos exames e conte as aprovações e reprovações

Imprima um resumo dos resultados do exame e decida se a mensalidade escolar deve ser elevada

Aqui, igualmente, embora tenhamos uma representação completa do programa inteiro, é necessário refinamento adicional. Agora empregamos variáveis específicas. Precisamos de contadores para registrar as aprovações e reprovações, de um contador para controlar o processo de loop e de uma variável para armazenar a entrada do usuário. A última variável não é inicializada, porque seu valor é lido a partir do usuário durante cada iteração do loop.

A instrução em pseudocódigo

Inicialize as variáveis

pode ser refinada desta maneira:

Inicialize as aprovações como zero

Inicialize as reprovações como zero

Inicialize o contador de alunos como um

Observe que somente os contadores são inicializados no início do algoritmo.

A instrução em pseudocódigo

Insira os 10 resultados dos exames e conte as aprovações e reprovações

requer um loop que sucessivamente insere o resultado de cada exame. Aqui sabemos com antecedência que há precisamente dez resultados de exame, desse modo o loop controlado por contador é apropriado. Dentro do loop (isto é, **aninhado** dentro do loop), uma instrução `if...else` se determinará se cada resultado de exame é uma passagem ou uma falha e incrementará o contador apropriado. O refinamento da instrução em pseudocódigo precedente é então

Enquanto o contador de alunos for menor ou igual a 10

Solicite que o usuário insira o próximo resultado de exame

Insira o próximo resultado de exame

Se o aluno foi aprovado

Adicione um a aprovações

Caso contrário

Adicione um a reprovações

Adicione um ao contador de alunos

Utilizamos linhas em branco para isolar a estrutura de controle *Se...Caso contrário (If...Else)*, que melhora a legibilidade.

A instrução em pseudocódigo

Imprima um resumo dos resultados do exame e decida se a mensalidade escolar deve ser elevada

pode ser refinada desta maneira:

Imprima o número de aprovações
Imprima o número de reprovações
Se mais de oito alunos forem aprovados
Imprima 'Aumentar a mensalidade escolar'

O segundo refinamento completo aparece na Figura 4.15. Observe que linhas em branco também são utilizadas para destacar a estrutura *Enquanto* para melhorar a legibilidade do programa. Esse pseudocódigo está agora suficientemente refinado para a conversão em C++.

Conversão em análise de classe

A classe C++ que implementa o algoritmo em pseudocódigo é mostrada na Figura 4.16–Figura 4.17 e duas execuções de exemplo aparecem na Figura 4.18.

As linhas 16–18 da Figura 4.17 declaram as variáveis que a função-membro `processExamResults` da classe `Analysis` utiliza para processar os resultados de teste. Observe que tiramos proveito de um recurso do C++ que permite que a inicialização da variável seja incorporada em declarações (passes [aprovações] é inicializada como 0, failures [reprovações] é inicializada como 0, e `studentCounter` é inicializada como 1). Programas de loop podem requerer inicialização no começo de cada repetição; essa reinicialização seria normalmente realizada por instruções de atribuição em vez de em declarações, ou movendo as declarações dentro dos corpos do loop.

A instrução `while` (linhas 22–36) itera dez vezes. Durante cada iteração, o loop insere e processa um dos resultados do exame. Observe que a instrução `if...else` (linhas 29–32) para processar cada resultado é aninhada na instrução `while`. Se o `result` for 1, a instrução `if...else` incrementa passes; caso contrário, ela assume que o `result` é 2 e incrementa failures.

```

1  Inicialize as aprovações como zero
2  Inicialize as reprovações como zero
3  Inicialize o contador de alunos como um
4
5  Enquanto o contador de alunos for menor ou igual a 10
6      Solicite que o usuário insira o próximo resultado de exame
7      Insira o próximo resultado de exame
8
9      Se o aluno foi aprovado
10         Adicione um a aprovações
11     Caso contrário
12         Adicione um a reprovações
13
14     Adicione um ao contador de alunos
15
16  Imprima o número de aprovações
17  Imprima o número de reprovações
18
19  Se mais de oito alunos forem aprovados
20     Imprima 'Aumentar a mensalidade escolar'
```

Figura 4.15 Pseudocódigo para o problema dos resultados do exame.

```

1  // Figura 4.16: Analysis.h
2  // Definição de análise de classe que analisa resultados de exame.
3  // A função-membro é definida em Analysis.cpp
4
5  // definição da classe Analysis
6  class Analysis
7  {
8  public:
9      void processExamResults(); // processa os resultados do teste de 10 alunos
10 }; // fim da classe Analysis
```

Figura 4.16 Problema dos resultados do exame: arquivo de cabeçalho `Analysis`.

```

1 // Figura 4.17: Analysis.cpp
2 // Definições de função-membro para a classe Analysis que
3 // analisa os resultados do teste.
4 #include <iostream>
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 // inclui a definição de classe Analysis a partir de Analysis.h
10 #include "Analysis.h"
11
12 // processa os resultados do teste de 10 alunos
13 void Analysis::processExamResults()
14 {
15     // inicializando variáveis nas declarações
16     int passes = 0; // número de aprovações
17     int failures = 0; // número de reprovações
18     int studentCounter = 1; // contador de alunos
19     int result; // o resultado de um teste (1 = aprovado, 2 = reprovado)
20
21     // processa 10 alunos utilizando o loop controlado por contador
22     while ( studentCounter <= 10 )
23     {
24         // solicita ao usuário uma entrada e obtém valor fornecido pelo usuário
25         cout << "Enter result (1 = pass, 2 = fail): ";
26         cin >> result; // lê o resultado
27
28         // if...else aninhado em while
29         if ( result == 1 ) // se resultado for 1,
30             passes = passes + 1; // incrementa aprovações;
31         else // senão resultado não for 1, então
32             failures = failures + 1; // incrementa reprovações
33
34         // incrementa studentCounter até o loop terminar
35         studentCounter = studentCounter + 1;
36     } // fim do while
37
38     // fase de terminação; exibe número de aprovados e reprovados
39     cout << "Passed " << passes << "\nFailed " << failures << endl;
40
41     // determina se mais de oito alunos passaram
42     if ( passes > 8 )
43         cout << "Raise tuition " << endl;
44 } // fim da função processExamResults

```

Figura 4.17 Problema dos resultados do exame: instruções de controle aninhados no arquivo de código-fonte Analysis.

A linha 35 incrementa `studentCounter` antes de a condição de loop ser testada novamente na linha 22. Depois que dez valores foram inseridos, o loop termina e a linha 39 exibe o número de passes e o número de failures. A instrução `if` nas linhas 42–43 determina se mais de oito alunos foram aprovados no exame e, se foram, gera saída da mensagem "Raise Tuition" (aumentar a mensalidade escolar).

*Demonstrando a classe **Analysis***

A Figura 4.18 cria um objeto `Analysis` (linha 7) e invoca a função-membro `processExamResults` do objeto (linha 8) para processar um conjunto de resultados de teste inserido pelo usuário. A Figura 4.18 mostra a entrada e a saída de duas execuções de exemplo do programa. No fim da primeira execução de exemplo, a condição na linha 42 da função-membro `processExamResults` na Figura 4.17 é verdadeira — mais de oito alunos passaram no teste, então o programa realiza saída de uma mensagem que indica que a matrícula deve ser feita.


```

1 // Figura 4.18: fig04_18.cpp
2 // Programa de teste para classe Analysis.
3 #include "Analysis.h" // inclui definição de classe Analysis
4
5 int main()
6 {
7     Analysis application; // cria o objeto da classe Analysis
8     application.processExamResults(); // função de chamada para processar resultados
9     return 0; // indica terminação bem-sucedida
10 } // fim de main

```

```

Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Passed 9
Failed 1
Raise tuition

```

```

Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 1
Enter result (1 = pass, 2 = fail): 2
Passed 6
Failed 4

```

Figura 4.18 Programa de teste para a classe Analysis.

4.1.1 Operadores de atribuição

O C++ fornece vários **operadores de atribuição** para abreviar expressões de atribuição. Por exemplo, a instrução

```
c = c + 3;
```

pode ser abreviada com o **operador de atribuição de adição +=** como

```
c += 3;
```

O operador += adiciona o valor da expressão à direita do operador ao valor da variável à esquerda do operador e armazena o resultado na variável à esquerda do operador. Qualquer instrução na forma

```
variável = variável operador expressão;
```

em que a mesma *variável* aparece em ambos os lados do *operador* de atribuição e o operador é um dos operadores binários +, -, *, / ou % (ou outros que discutiremos mais adiante no texto), pode ser escrita na forma

```
variável operador = expressão;
```

Assim a atribuição `c += 3` adiciona 3 a c. A Figura 4.19 mostra expressões de exemplo de operadores de atribuição aritméticos que utilizam esses operadores e suas explicações.

Operador de atribuição	Expressão de exemplo	Explicação	Atribuições
Assuma: <code>int c = 3, d = 5, e = 4, f = 6, g = 12;</code>			
<code>+=</code>	<code>c += 7</code>	<code>c = c + 7</code>	10 a c
<code>-=</code>	<code>d -= 4</code>	<code>d = d - 4</code>	1 a d
<code>*=</code>	<code>e *= 5</code>	<code>e = e * 5</code>	20 a e
<code>/=</code>	<code>f /= 3</code>	<code>f = f / 3</code>	2 a f
<code>%=</code>	<code>g %= 9</code>	<code>g = g % 9</code>	3 a g

Figura 4.19 Operadores de atribuição aritméticos.

4.12 Operadores de incremento e decremento

Além dos operadores de atribuição aritméticos, o C++ também fornece dois operadores unários para adicionar 1 ao ou subtrair 1 do valor de uma variável numérica. Esses são o **operador de incremento** unário, `++`, e o **operador de decremento** unário, `--`, que são resumidos na Figura 4.20. Um programa pode incrementar por 1 o valor de uma variável chamada `c` utilizando o operador de incremento, `++`, em vez da expressão `c = c + 1` ou `c += 1`. Um operador de incremento ou decremento que é prefixado a (colocado antes de) uma variável é referido como **operador de incremento prefixado** ou **operador de decremento prefixado**, respectivamente. Um operador de incremento ou de decremento que é colocado depois de uma variável é chamado **operador de pós-incremento** ou **operador de pós-decremento**, respectivamente.

Utilizar o operador de pré-incremento (ou de pré-decremento) para adicionar (ou subtrair) 1 de uma variável é conhecido como **pré-incrementar** (ou **pré-decrementar**) a variável. Pré-incrementar (ou pré-decrementar) faz com que a variável seja incrementada (decrementada) por 1, e então o novo valor da variável é utilizado na expressão em que ela aparece. Utilizar o operador de pós-incremento (ou pós-decremento) para adicionar (ou subtrair) 1 de uma variável é conhecido como **pós-incrementar** (ou **pós-decrementar**) a variável. Pós-incrementar (pós-decrementar) faz com que o valor atual da variável seja utilizado na expressão em que ela aparece, e então o valor da variável é incrementado (decrementado) por 1.



Boa prática de programação 4.8

Diferentemente dos operadores binários, os operadores de incremento e decremento unários devem ser colocados ao lado dos seus operandos, sem espaços no meio.

A Figura 4.21 demonstra a diferença entre as versões de pré-incremento e de pós-incremento do operador de incremento `++`. O operador de decremento (`--`) funciona de maneira semelhante. Observe que esse exemplo não contém uma classe, mas apenas um arquivo de código-fonte com a função `main` que realiza todo o trabalho do aplicativo. Neste capítulo e no Capítulo 3, você viu exemplos consistindo em uma classe (incluindo o cabeçalho e arquivos de código-fonte para essa classe), bem como outro arquivo de código-fonte que testa a classe. Esse arquivo de código-fonte continha a função `main`, que criava um objeto da classe e chamava suas funções-membro. Neste exemplo, simplesmente queremos mostrar os mecanismos do operador `++`, e, então, utilizar apenas um arquivo de código-fonte com a função `main`. Ocasionalmente, quando não faz sentido tentar criar uma classe reutilizável para demonstrar um conceito simples, utilizaremos um exemplo mecânico contido inteiramente dentro da função `main` de um único arquivo de código-fonte.

Operador	Chamado	Expressão de exemplo	Explicação
<code>++</code>	pré-incremento	<code>++a</code>	Incremente <code>a</code> por 1, então utilize o novo valor de <code>a</code> na expressão em que <code>a</code> reside.
<code>++</code>	pós-incremento	<code>a++</code>	Utilize o valor atual de <code>a</code> na expressão em que <code>a</code> reside, então incremente <code>a</code> por 1.
<code>--</code>	pré-decremento	<code>--b</code>	Decremente <code>b</code> por 1, então utilize o novo valor de <code>b</code> na expressão em que <code>b</code> reside.
<code>--</code>	pós-decremento	<code>b--</code>	Utilize o valor atual de <code>b</code> na expressão em que <code>b</code> reside, então decremente <code>b</code> por 1.

Figura 4.20 Operadores de incremento e de decremento.

```

1 // Figura 4.21: fig04_21.cpp
2 // Pré-incrementando e pós-incrementando.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     int c;
10
11     // demonstra pós-incremento
12     c = 5; // atribui 5 à variável c
13     cout << c << endl; // imprime 5
14     cout << c++ << endl; // imprime 5 então pós-incrementa
15     cout << c << endl; // imprime 6
16
17     cout << endl; // pula uma linha
18
19     // demonstra pré-incremento
20     c = 5; // atribui 5 à variável c
21     cout << c << endl; // imprime 5
22     cout << ++c << endl; // pré-incrementa e então imprime 6
23     cout << c << endl; // imprime 6
24     return 0; // indica terminação bem-sucedida
25 } // fim de main

```

```

5
5
6

5
6
6

```

Figura 4.21 Pré-incrementando e pós-incrementando.

A linha 12 inicializa a variável `c` como 5 e a linha 13 gera a saída do valor inicial da `c`. A linha 14 gera a saída do valor da expressão `c++`. Essa expressão pós-incrementa a variável `c`, assim o valor original de `c` (5) é enviado para a saída, e então o valor de `c` é incrementado. Portanto, a linha 14 gera a saída do valor inicial de `c` (5) novamente. A linha 15 gera a saída do novo valor de `c` (6) para provar que o valor da variável foi de fato incrementado na linha 14.

A linha 20 reinicializa o valor da `c` como 5 e a linha 21 envia o valor de `c` para a saída. A linha 22 gera a saída do valor da expressão `++c`. Essa expressão pré-incrementa `c`, dessa forma seu valor é incrementado e então o novo valor (6) é enviado para a saída. A linha 23 gera a saída do valor de `c` novamente para mostrar que o valor de `c` ainda é 6 depois que a linha 22 é executada.

Os operadores aritméticos de atribuição e os operadores de incremento e decremento podem ser utilizados para simplificar as instruções de um programa. As três instruções de atribuição na Figura 4.17

```

pases = pases + 1;
failures = failures + 1;
studentCounter = studentCounter + 1;

```

podem ser escritas mais concisamente com operadores de atribuição como

```

pases += 1;
failures += 1;
studentCounter += 1;

```

com operadores de pré-incremento como

```

++pases;
++failures;
++studentCounter;

```

ou com operadores de pós-incremento como

```
passes++;
failures++;
studentCounter++;
```

Observe que, quando a incrementação (++) ou decrementação (--) de uma variável ocorre em uma instrução por si mesma, as formas de pré-incremento e de pós-incremento e as formas de pré-decremento e pós-decremento têm o mesmo efeito. Somente quando uma variável aparece no contexto de uma expressão maior é que pré-incrementar e pós-incrementar a variável têm efeitos diferentes (e igualmente para pré-decrementar e pós-decrementar).



Erro comum de programação 4.14

Tentar utilizar o operador de incremento ou decremento em uma expressão diferente da de um nome variável ou referência modificável, por exemplo, escrever `++(x + 1)`, é um erro de sintaxe.

A Figura 4.22 mostra a precedência e a associatividade dos operadores introduzidos nesse ponto. Os operadores são mostrados de cima para baixo em ordem decrescente de precedência. A segunda coluna indica a associatividade dos operadores em cada nível de precedência. Note que o operador condicional (`?:`), os operadores unários de pré-incremento (++) , pré-decremento (--), adição (+) e subtração (-), e os operadores de atribuição =, +=, -=, *=, /= e %= associam-se da direita para a esquerda. Todos os outros operadores no gráfico de precedência de operador da Figura 4.22 são associados da esquerda para a direita. A terceira coluna nomeia os vários grupos de operadores.

4.13 Estudo de caso de engenharia de software: identificando atributos de classe no sistema ATM (opcional)

Na Seção 3.11, começamos a primeira etapa de um projeto orientado a objetos (*object-oriented project* – OOD) para nosso sistema ATM — analisando o documento de requisitos e identificando as classes necessárias para implementar o sistema. Listamos os substantivos e frases com substantivos no documento de requisitos e identificamos uma classe separada para cada uma que desempenha um papel significativo no sistema ATM. Em seguida, modelamos as classes e seus relacionamentos em um diagrama de classes UML (Figura 3.23). As classes contêm atributos (dados) e operações (comportamentos). Os atributos de classe são implementados em programas C++ como membros de dados; e as operações de classe são implementadas como funções-membro. Nesta seção, determinamos muitos dos atributos necessários no sistema ATM. No Capítulo 5, examinaremos como esses atributos representam o estado de um objeto. No Capítulo 6, determinaremos as operações de classe.

Identificando atributos

Considere os atributos de alguns objetos do mundo real: os atributos de uma pessoa incluem a altura, peso e se a pessoa é canhota, destra ou ambidestra. Os atributos de um rádio incluem sua configuração de estações, configuração de volume e configuração de AM ou FM. Os atributos de um carro incluem as leituras do velocímetro e do odômetro, a quantidade de gasolina no tanque e a marcha em que ele

Operadores	Associatividade	Tipo
()	da esquerda para a direita	parênteses
++ -- <code>static_cast<tipo>()</code>	da esquerda para a direita	unário (pós-fixado)
++ -- + -	da direita para a esquerda	unário (prefixo)
* / %	da esquerda para a direita	multiplicativo
+ -	da esquerda para a direita	aditivo
<< >>	da esquerda para a direita	inserção/extração
< <= > >=	da esquerda para a direita	relacional
== !=	da esquerda para a direita	igualdade
?:	da direita para a esquerda	ternário condicional
= += -= *= /= %=	da direita para a esquerda	atribuição

Figura 4.22 Ordem de precedência dos operadores encontrados até agora no texto.

está. Os atributos de um computador pessoal incluem seu fabricante (por exemplo, Dell, Sun, Apple ou IBM), tipo de tela (por exemplo, LCD ou CRT), tamanho da memória principal e tamanho do disco rígido.

Podemos identificar muitos atributos das classes no nosso sistema procurando palavras e frases descritivas no documento de requisitos. Para cada palavra ou frase encontrada que desempenha um papel significativo no sistema ATM, criamos um atributo e o atribuímos a uma ou mais classes identificadas na Seção 3.11. Também criamos atributos para representar quaisquer dados adicionais de que uma classe talvez precise, à medida que essas necessidades se tornam claras por todo o processo do projeto.

A Figura 4.23 lista as palavras ou frases no documento de requisitos que descrevem cada classe. Formamos essa lista lendo o documento de requisitos e identificando todas as palavras ou frases que se referem às características das classes no sistema. Por exemplo, o documento de requisitos descreve os passos seguidos para obter uma ‘quantia de saque’ (*‘withdrawal amount’*), assim listamos *‘amount’* ao lado da classe `Withdrawal`.

A Figura 4.23 nos leva a criar um atributo da classe ATM. A classe ATM mantém as informações sobre o estado do ATM. A frase ‘usuário é autenticado’ descreve um estado do ATM (introduzimos estados na Seção 5.11), portanto incluímos `userAuthenticated` como um **atributo Boolean** (isto é, um atributo com um valor `true` ou `false`). O tipo UML `Boolean` é equivalente ao tipo `bool` em C++. Esse atributo indica se o ATM autenticou o usuário atual com sucesso — `userAuthenticated` dever ser `true` para que o sistema autorize o usuário a realizar transações e acessar as informações sobre a conta. Esse atributo ajuda a garantir a segurança dos dados no sistema.

As classes `BalanceInquiry`, `Withdrawal` e `Deposit` compartilham um atributo. Cada transação envolve um ‘número de conta’ que corresponde à conta do usuário que faz a transação. Atribuímos um atributo inteiro `accountNumber` a cada classe de transação para identificar a conta a qual um objeto da classe se aplica.

Palavras e frases descritivas no documento de requisitos também sugerem algumas diferenças nos atributos requeridos em cada classe de transação. O documento de requisitos indica que, para sacar dinheiro ou depositar fundos, os usuários devem inserir uma ‘quantia’ (*‘amount’*) específica a ser sacada ou depositada, respectivamente. Portanto, atribuímos às classes `Withdrawal` e `Deposit` um atributo `amount` para armazenar o valor fornecido pelo usuário. Os valores relacionados a um saque e depósito são características definidoras das transações que o sistema requer para que essas transações aconteçam. A classe `BalanceInquiry`, porém, não precisa de nenhum dado adicional para realizar sua tarefa — ela requer somente um número de conta para indicar a conta cujo saldo deve ser recuperado.

A classe `Account` tem vários atributos. O documento de requisitos declara que cada conta bancária deve ter um ‘número de conta’ e um ‘PIN’ que o sistema utiliza para identificar contas e autenticar usuários. Atribuímos à classe `Account` dois atributos de inteiro: `accountNumber` e `pin`. O documento de requisitos também especifica que uma conta deve manter um ‘saldo’ (*‘balance’*) do valor na conta e que esse valor depositado pelo usuário não se torna disponível para um saque até que o banco o verifique no envelope de depósito e que todos os cheques no envelope sejam compensados. Uma conta, porém, ainda deve registrar o valor que um usuário deposita. Portanto, decidimos que uma conta deve representar um saldo que utiliza dois atributos de tipo UML `Double`: `availableBalance` e `totalBalance`. O atributo `availableBalance` armazena o valor que um usuário pode sacar da conta. O atributo `totalBalance` refere-se ao valor total que o usuário tem ‘em depósito’ (isto é, o valor disponível mais o valor esperando para ser verificado ou compensado). Por exemplo,

Classe	Palavras e frases descritivas
ATM	usuário é autenticado
BalanceInquiry	número de conta
Withdrawal	número de conta valor
Deposit	número de conta valor
BankDatabase	[nenhuma palavra ou frase descritiva]
Account	número de conta PIN saldo
Screen	[nenhuma palavra ou frase descritiva]
Keypad	[nenhuma palavra ou frase descritiva]
CashDispenser	inicia cada dia carregado com 500 cédulas de \$ 20
DepositSlot	[nenhuma palavra ou frase descritiva]

Figura 4.23 Palavras e frases descritivas nos requisitos do ATM.

suponha que um usuário do ATM deposite \$ 50,00 em uma conta zerada. O atributo `totalBalance` aumentaria para \$ 50,00 a fim de registrar o depósito, mas o `availableBalance` permaneceria em \$ 0. [Nota: Supomos que o banco atualiza o atributo `availableBalance` de uma `Account` logo depois que a transação ATM ocorre, em resposta à confirmação de que o valor de \$ 50,00 em dinheiro ou cheque foi encontrado no envelope de depósito. Assumimos que essa atualização ocorre por meio de uma transação que um funcionário do banco realiza utilizando algum software do banco diferente do ATM. Portanto, não discutimos essa transação no nosso estudo de caso.]

A classe `CashDispenser` tem um atributo. O documento de requisitos afirma que o dispensador de cédulas ‘começa todos os dias carregado com 500 cédulas de \$ 20’. O dispensador de cédulas deve manter um registro do número de cédulas que ele contém para determinar se há dinheiro suficiente à disposição para satisfazer as solicitações de saque. Atribuímos à classe `CashDispenser` um atributo inteiro `count`, inicialmente configurado como 500.

Para problemas reais na indústria, não há garantias de que os documentos de requisitos serão suficientemente detalhados e precisos para que o projetista de sistemas orientados a objetos determine todos os atributos ou mesmo todas as classes. A necessidade de classes, atributos e comportamentos adicionais pode tornar-se clara à medida que o projeto avança. À medida que progredimos por esse estudo de caso, também continuaremos a adicionar, modificar e excluir as informações sobre as classes no nosso sistema.

Modelando atributos

O diagrama de classes na Figura 4.24 lista alguns atributos das classes no nosso sistema — as palavras e frases descritivas na Figura 4.23 nos ajudaram a identificar esses atributos. Por simplicidade, a Figura 4.24 não mostra as associações entre as classes — estas são mostradas na Figura 3.23. Essa é uma prática comum entre projetistas de sistemas ao desenvolver projetos. Lembre-se, na Seção 3.11, de que na UML os atributos de uma classe são colocados no compartimento central do retângulo da classe. Listamos cada nome e tipo do atributo separado por dois-pontos (:), seguido, em alguns casos, por um sinal de igual (=) e de um valor inicial.

Considere o atributo `userAuthenticated` da classe `ATM`:

```
userAuthenticated : Boolean = false
```

Essa declaração de atributo contém três informações sobre o atributo. O **nome do atributo** é `userAuthenticated`. O **tipo do atributo** é `Boolean`. Em C++, um atributo pode ser representado por um tipo fundamental, como o tipo `bool`, `int` ou `double` ou um tipo de classe — como discutido no Capítulo 3. Escolhemos modelar apenas atributos de tipo primitivo na Figura 4.24 — discutimos o raciocínio por trás dessa decisão em breve. [Nota: A Figura 4.24 lista os tipos de dados UML para os atributos. Quando implementarmos o sistema, associaremos os tipos UML `Boolean`, `Integer` e `Double` com os tipos fundamentais do C++ `bool`, `int` e `double`, respectivamente.]

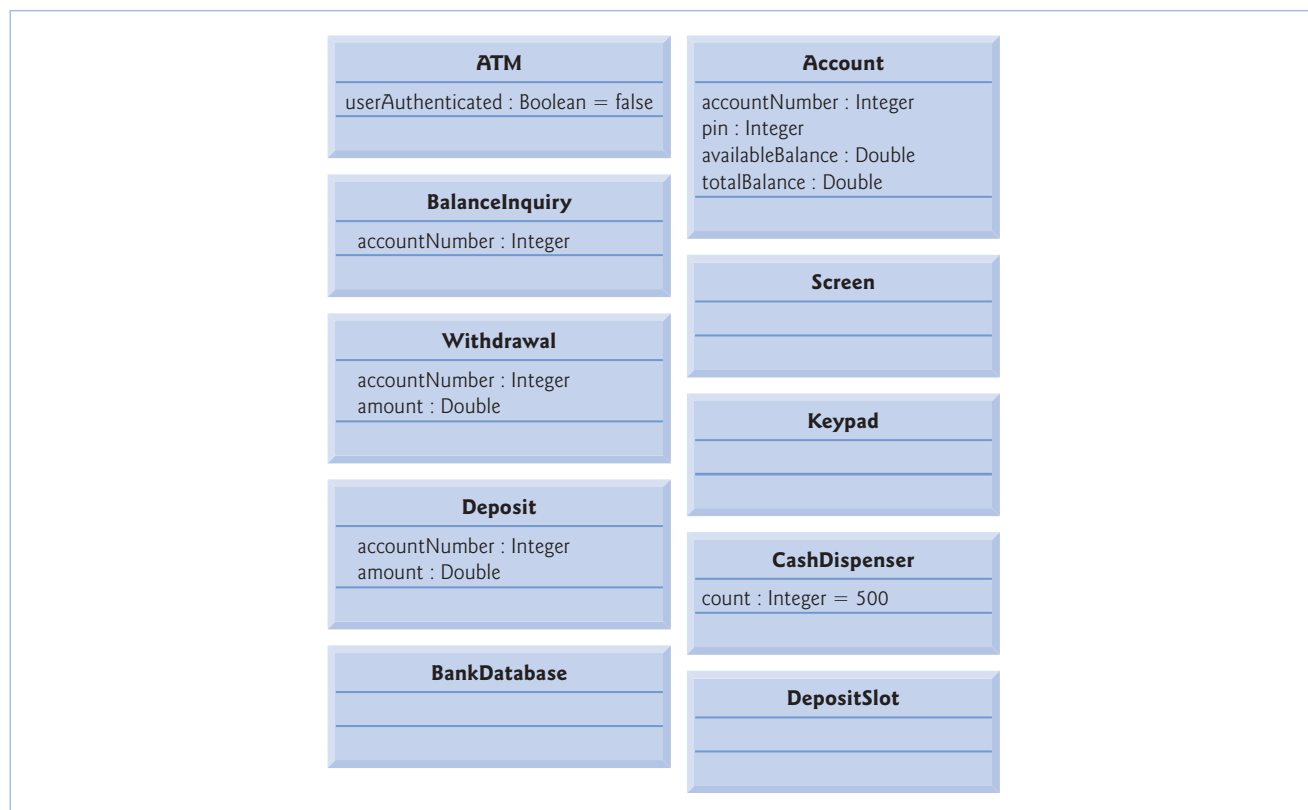


Figura 4.24 Classes com atributos.

Também podemos indicar um valor inicial para um atributo. O atributo `userAuthenticated` na classe `ATM` tem um valor inicial `false`. Isso indica que o sistema inicialmente não considera o usuário como autenticado. Se um atributo não contiver nenhum valor inicial especificado, somente seu nome e tipo (separado por dois-pontos) são mostrados. Por exemplo, o atributo `accountNumber` da classe `BalanceInquiry` é um `Integer`. Aqui mostramos o valor não inicial, porque o valor desse atributo é um número que ainda não conhecemos. Esse número será determinado em tempo de execução com base no número de conta inserido pelo usuário atual do `ATM`.

A Figura 4.24 não inclui nenhum atributo para as classes `Screen`, `Keypad` e `DepositSlot`. Estas são componentes importantes do nosso sistema, para as quais o processo do nosso projeto simplesmente ainda não revelou nenhum atributo. Entretanto, ainda é possível descobrir algumas nas fases restantes do projeto ou ao implementarmos essas classes em C++. Isso é perfeitamente normal para o processo iterativo de engenharia de software.



Observação de engenharia de software 4.8

Nas etapas iniciais do processo de projeto, freqüentemente faltam nas classes atributos (e operações). Essas classes, porém, não devem ser eliminadas, pois os atributos (e operações) podem tornar-se evidentes nas fases posteriores do projeto e implementação.

Observe que a Figura 4.24 também não inclui nenhum atributo para a classe `BankDatabase`. Considerando a discussão no Capítulo 3, lembre-se de que em C++ os atributos podem ser representados tanto por tipos fundamentais como por tipos de classe. Optamos por incluir somente os atributos de tipo fundamental ao diagrama de classes na Figura 4.24 (e aos diagramas de classes semelhantes por todo o estudo de caso). Um atributo de tipo de classe é modelado mais claramente como uma associação (em particular, uma composição) entre a classe com o atributo e a classe do objeto do qual o atributo é uma instância. Por exemplo, o diagrama de classes na Figura 3.23 indica que a classe `BankDatabase` participa de um relacionamento de composição com zero ou mais objetos `Account`. A partir dessa composição, podemos determinar que, ao implementarmos o sistema `ATM` em C++, será necessário criar um atributo da classe `BankDatabase` para armazenar zero ou mais objetos de `Account`. De modo semelhante, atribuiremos os atributos à classe `ATM` que corresponde a seus relacionamentos de composição com classes `Screen`, `Keypad`, `CashDispenser` e `DepositSlot`. Esses atributos baseados em composição seriam redundantes se modelados na Figura 4.24, porque as composições modeladas na Figura 3.23 já comunicam o fato de que o banco de dados contém as informações sobre zero ou mais contas e que um `ATM` é composto de uma tela, teclado, dispensador de cédulas e uma abertura para depósito. Em geral, os desenvolvedores de software modelam esses relacionamentos integrais/parciais como composições em vez de como atributos requeridos para implementar os relacionamentos.

O diagrama de classes na Figura 4.24 fornece uma base sólida para a estrutura do nosso modelo, mas o diagrama não está completo. Na Seção 5.11, identificamos os estados e atividades dos objetos no modelo e na Seção 6.22 identificamos as operações que os objetos realizam. À medida que apresentarmos outras informações sobre a UML e o projeto orientado a objetos, continuaremos a fortalecer a estrutura do nosso modelo.

Exercícios de revisão do estudo de caso de engenharia de software

- 4.1** Em geral, identificamos os atributos das classes no nosso sistema analisando os(as) _____ no documento de requisitos.
- substantivos simples e substantivos compostos
 - palavras e frases descritivas
 - verbos e frases com verbos
 - Todos os acima.
- 4.2** Qual dos seguintes não é um atributo de um avião?
- comprimento
 - envergadura da asa
 - vôo
 - número de poltronas
- 4.3** Descreva o significado da seguinte declaração de atributo da classe `CashDispenser` no diagrama de classes na Figura 4.24:
- ```
count : Integer = 500
```

### Respostas aos exercícios de revisão do estudo de caso de engenharia de software

- 4.1** b.
- 4.2** c. Vôo é uma operação ou comportamento de um avião, não um atributo.
- 4.3** Isso indica que `count` é um `Integer` com um valor inicial de 500. Esse atributo monitora o número de notas disponível na `CashDispenser` em determinado momento.

## 4.14 Síntese

Este capítulo apresentou técnicas básicas de solução de problemas que programadores utilizam na criação de classes e desenvolvimento de funções-membro para essas classes. Demonstramos como construir um algoritmo (isto é, uma abordagem para resolver um problema) e então como refinar o algoritmo por meio de várias fases de desenvolvimento em pseudocódigo, resultando em código C++ que pode

ser executado como parte de uma função-membro. Você aprendeu a utilizar o refinamento passo a passo de cima para baixo para planejar as ações específicas que uma função deve realizar e a ordem em que ela deve realizá-las.

Você aprendeu que apenas três tipos de estruturas de controle — sequência, seleção e repetição — são necessários para desenvolver qualquer algoritmo. Demonstramos duas das instruções de seleção do C++ — a instrução de uma única seleção `if` e a instrução de seleção dupla `if...else`. A instrução `if` é utilizada para executar um conjunto de instruções baseadas em uma condição — se a condição for verdadeira, as instruções são executadas; se não, as instruções são ignoradas. A instrução de seleção dupla `if...else` é utilizada para executar um conjunto de instruções se uma condição for verdadeira, e outro conjunto de instruções se a condição for falsa. Discutimos então a instrução de repetição `while`, onde um conjunto de instruções é executado repetidamente se uma condição for verdadeira. Utilizamos o empilhamento de instruções de controle para totalizar e calcular a média de um conjunto de notas de alunos com a repetição controlada por contador e por sentinela e utilizamos o aninhamento de instruções de controle para analisar e tomar decisões com base em um conjunto de resultados de um exame. Introduzimos operadores de atribuição, que podem ser utilizados para abreviar instruções. Apresentamos os operadores de incremento e de decremento, que podem ser utilizados para adicionar ou subtrair o valor 1 de uma variável. No Capítulo 5, “Instruções de controle: parte 2”, continuamos nossa discussão sobre as instruções de controle, introduzindo as instruções `for`, `do...while` e `switch`.

## Resumo

- Um algoritmo é um procedimento para resolver um problema em termos das ações a executar e a ordem em que executá-las.
- Especificar a ordem em que as instruções (ações) são executadas em um programa é chamado controle de programa.
- O pseudocódigo ajuda um programador a pensar sobre um programa antes de tentar escrevê-lo em uma linguagem de programação.
- Os diagramas de atividades são parte da Unified Modeling Language (UML) — um padrão da indústria para modelagem de sistemas de software.
- Um diagrama de atividades modela o fluxo de trabalho (também chamado atividade) de um sistema de software.
- Os diagramas de atividades são compostos de símbolos de uso especial, como símbolos do estado da ação, losangos e pequenos círculos. Esses símbolos são conectados por setas de transição que representam o fluxo da atividade.
- Como ocorre com o pseudocódigo, os diagramas de atividades ajudam os programadores a desenvolver e representar algoritmos.
- O estado de uma ação é representado por um retângulo com seus lados esquerdo e direito substituídos por arcos curvados para fora. A expressão da ação aparece dentro do estado da ação.
- As setas em um diagrama de atividades representam transições, as quais indicam a ordem em que ocorrem as ações representadas pelos estados da ação.
- O círculo sólido localizado na parte superior de um diagrama de atividades representa o estado inicial — o começo do fluxo de trabalho antes de o programa realizar as ações modeladas.
- O círculo sólido cercado por um círculo vazio que aparece na parte inferior do diagrama de atividades representa o estado final — o fim do fluxo de trabalho depois que o programa realiza suas ações.
- Retângulos com o canto superior direito dobrado chamam-se notas na UML. As notas são observações explanatórias que descrevem o propósito dos símbolos no diagrama. Uma linha pontilhada conecta cada nota ao elemento que a nota descreve.
- Um losango ou símbolo de decisão em um diagrama de atividades indica que uma decisão deve ser tomada. O fluxo de trabalho continuará ao longo de um caminho determinado pelas condições de guarda do símbolo associado, que podem ser verdadeiras ou falsas. Cada seta de transição que sai de um símbolo de decisão tem uma condição de guarda (especificada entre colchetes ao lado da seta de transição). Se uma condição de guarda for verdadeira, o fluxo de trabalho entra no estado de ação para o qual a seta de transição aponta.
- Um losango em um diagrama de atividades também representa o símbolo de agregação, que une dois fluxos de atividade em um. Um símbolo de agregação contém duas ou mais setas de transição apontando para o losango e somente uma seta de transição apontando a partir do losango, para indicar a conexão de múltiplos fluxos de atividades a fim de continuar a atividade.
- O refinamento passo a passo de cima para baixo é um processo para refinar o pseudocódigo mantendo uma representação completa do programa durante cada refinamento.
- Há três tipos de estruturas de controle — sequência, seleção e repetição.
- A estrutura de sequência é parte integrante do C++ — por padrão, as instruções são executadas na ordem em que elas aparecem.
- Uma estrutura de seleção escolhe entre cursos alternativos da ação.
- A instrução de uma única seleção `if` realiza (seleciona) uma ação se uma condição for verdadeira ou ignora a ação se a condição for falsa.
- A instrução de seleção dupla `if...else` realiza (seleciona) uma ação se uma condição for verdadeira e realiza uma ação diferente se a condição for falsa.

- Para incluir várias instruções no corpo de um `if` (ou no corpo de um `else` para uma instrução `if...else`), inclua as instruções dentro de chaves (`{ e }`). Um conjunto de instruções contidas dentro de um par de chaves é chamado bloco. Um bloco pode ser colocado em qualquer lugar em um programa em que uma única instrução pode ser colocada.
- Uma instrução nula, indicando que nenhuma ação deve ser tomada, é indicada por um ponto-e-vírgula (`;`).
- Uma instrução repetição específica que uma ação deve ser repetida enquanto algumas condições permanecem verdadeiras.
- Um valor que contém uma parte fracionária é referido como um número de ponto flutuante e é representado aproximadamente por tipos de dados como `float` e `double`.
- A repetição controlada por contador é utilizada quando o número de repetições é conhecido antes de um loop começar a executar, isto é, quando há repetição definida.
- O operador de coerção unário `static_cast` pode ser utilizado para criar uma cópia de ponto flutuante temporária de seu operando.
- Os operadores unários aceitam apenas um operando; operadores binários aceitam dois.
- O manipulador de fluxo parametrizado `setprecision` indica o número de dígitos de precisão que deve ser exibido à direita do ponto de fração decimal.
- O manipulador de fluxo `fixed` indica que a saída dos valores de ponto flutuante deve ser no chamado formato de ponto fixo, em oposição à notação científica.
- A repetição controlada por sentinela é utilizada quando o número de repetições não é conhecido antes de um loop começar a executar, isto é, quando há repetição indefinida.
- Uma instrução de controle aninhada aparece no corpo de outra instrução de controle.
- O C++ fornece os operadores de atribuição aritméticos `+=`, `-=`, `*=`, `/=` e `%=` para abreviar expressões de atribuição.
- O operador de incremento, `++`, e o operador de decremento, `--`, incrementam ou decrementam uma variável por 1, respectivamente. Se o operador for prefixado à variável, a variável é primeiro incrementada ou decrementada por 1 e então seu novo valor é utilizado na expressão em que ela aparece. Se o operador for pós-fixado à variável, a variável é primeiro utilizada na expressão em que aparece e então o valor da variável é incrementado ou é decrementado por 1.

## Terminologia

|                                                                   |                                                      |                                                      |
|-------------------------------------------------------------------|------------------------------------------------------|------------------------------------------------------|
| ação                                                              | erro, <i>off-by-one</i>                              | iterar                                               |
| 'bombing'                                                         | estado da ação                                       | linha pontilhada                                     |
| 'crashing'                                                        | estado final                                         | loop                                                 |
| algoritmo                                                         | estado inicial                                       | loop aninhado dentro de um loop                      |
| aninhamento de instruções de controle                             | execução sequencial                                  | loop, condição de continuação                        |
| aproximação de números de ponto flutuante                         | expressão condicional                                | losango, símbolo                                     |
| arredondando                                                      | expressão de ação                                    | manipulador de fluxo                                 |
| associar da direita para a esquerda                               | fluxo de trabalho de parte de um sistema de software | manipulador de fluxo <code>fixed</code>              |
| associar da esquerda para a direita                               | formato de ponto fixo                                | manipulador de fluxo não parametrizado               |
| bloco                                                             | goto, instrução                                      | manipulador de fluxo parametrizado                   |
| bool                                                              | incremento, operador <code>++</code>                 | manipulador de fluxo <code>setprecision</code>       |
| cálculo de média                                                  | instrução composta                                   | manipulador de fluxo <code>showpoint</code>          |
| constante de ponto flutuante                                      | instrução de controle                                | modelo de programação de ação/decisão                |
| contador                                                          | instrução de controle aninhado                       | nota                                                 |
| controle de programa                                              | instrução de controle de entrada única/saída única   | notação científica                                   |
| conversão explícita                                               | instrução de dupla seleção <code>if...else</code>    | número de ponto flutuante de precisão dupla          |
| conversão implícita                                               | instrução de loop                                    | número de ponto flutuante de precisão simples        |
| decremento, operador <code>--</code>                              | instrução de múltipla seleção                        | operador aritmético binário                          |
| design orientado a objetos ( <i>object-oriented design</i> – OOD) | instrução de seleção dupla                           | operador condicional ( <code>?:</code> )             |
| diagrama de atividades                                            | instrução de uma única seleção <code>if</code> ,     | operador de atribuição de adição ( <code>+=</code> ) |
| diagrama de atividades de instrução de sequência                  | instrução executável                                 | operador de coerção                                  |
| divisão de inteiro                                                | instrução nula                                       | operador de pós-decremento                           |
| eliminação de goto                                                | instrução vazia                                      | operador de pós-incremento                           |
| empilhamento de instruções de controle                            | iterações de loop                                    | operador de pré-decremento                           |
| erro fatal de lógica                                              | iterações de um loop                                 | operador de pré-incremento                           |
|                                                                   |                                                      | operador ternário                                    |
|                                                                   |                                                      | operador unário                                      |

|                                            |                                              |                                             |
|--------------------------------------------|----------------------------------------------|---------------------------------------------|
| operador unário de adição (+)              | procedimento                                 | símbolo de intercalação                     |
| operador unário de coerção                 | programação estruturada                      | símbolo de seta                             |
| operador unário de subtração (-)           | promoção                                     | símbolo de seta de transição                |
| operadores de atribuição                   | promoção de inteiro                          | tipo de dados <code>double</code>           |
| operadores de atribuição aritméticos       | pseudocódigo                                 | tipo de dados <code>float</code>            |
| operando                                   | refinamento passo a passo de cima para baixo | topo                                        |
| ordem em que as ações devem ser executadas | repetição controlada por contador            | total                                       |
| palavras-chave                             | repetição controlado por sentinela           | transferência de controle                   |
| ponto flutuante, número                    | repetição definida                           | transição                                   |
| pós-decremento                             | repetição indefinida                         | truncar                                     |
| pós-incremento                             | repetição, instrução                         | valor "lixo"                                |
| precedência de operadores                  | segundo refinamento                          | valor de flag                               |
| precisão                                   | seleção, instrução                           | valor de sentinela                          |
| precisão-padrão                            | seqüência, instrução                         | valor de sinal                              |
| pré-decremento                             | símbolo da ação de estado                    | valor fictício                              |
| pré-incremento                             | símbolo de círculo pequeno                   | valor indefinido                            |
| primeiro refinamento                       | símbolo de círculo sólido                    | <code>while</code> , instrução de repetição |
| problema do <code>else</code> oscilante    | símbolo de decisão                           |                                             |

## Exercícios de revisão

**4.1** Responda cada uma das seguintes perguntas.

- Todos os programas podem ser escritos em termos de três tipos de estruturas de controle: \_\_\_\_\_, \_\_\_\_\_ e \_\_\_\_\_.
- A instrução de seleção \_\_\_\_\_ é utilizada para executar uma ação quando uma condição é `true` ou uma ação diferente quando essa condição for `false`.
- Repetir um conjunto de instruções por um número específico de vezes é chamado de repetição \_\_\_\_\_.
- Quando não se sabe antecipadamente quantas vezes um conjunto de instruções será repetido, um valor \_\_\_\_\_ pode ser utilizado para terminar a repetição.

**4.2** Escreva quatro instruções C++ diferentes que adicionam 1 à variável do tipo inteiro `x`.

**4.3** Escreva instruções C++ para realizar cada uma das seguintes tarefas:

- Em uma instrução, atribua a soma do valor atual de `x` e `y` a `z` e pós-incremento o valor de `x`.
- Determine se o valor da variável `count` é maior que 10. Se for, imprima "Count is greater than 10".
- Pré-decremente a variável `x` por 1, então subtraia o resultado da variável `total`.
- Calcule o resto após `q` ser dividido pelo `divisor` e atribua o resultado a `q`. Escreva essa instrução de duas maneiras diferentes.

**4.4** Escreva instruções C++ para realizar cada uma das seguintes tarefas.

- Declare variáveis `sum` e `x` que serão de tipo `int`.
- Configure a variável `x` como 1.
- Configure a variável `sum` como 0.
- Adicione variável `x` à variável `sum` e atribua o resultado à variável `sum`.
- Imprima "The sum is: " seguido pelo valor da variável `sum`.

**4.5** Combine as instruções escritas no Exercício 4.4 em um programa em C++ que calcula e imprime a soma dos inteiros de 1 a 10. Utilize a estrutura `while` para fazer loop pelas instruções de cálculo e incremento. O loop deve terminar quando o valor de `x` se tornar 11.

**4.6** Mostre os valores de cada variável depois que o cálculo é realizado. Assuma que quando cada instrução começa a executar, todas as variáveis têm o valor inteiro 5.

- `product *= x++;`
- `quotient /= ++x;`

**4.7** Escreva instruções C++ únicas que realizem o seguinte:

- Insiram a variável de inteiro `x` com `cin >>`.
- Insiram a variável de inteiro `y` com `cin >>`.
- Configurem variável de inteiro `i` como 1.
- Configurem variável de inteiro `power` como 1.
- Multipliquem a variável `power` por `x` e atribuam o resultado a `power`.
- Pós-incrementem variável `i` por 1.
- Determinem se `i` é menor que ou igual a `y`.
- Realizem saída da variável de inteiro `power` com `cout <<`.

**4.8** Escreva um programa C++ que utiliza as instruções do Exercício 4.7 para calcular  $x$  elevado a  $y$  potência. O programa deve ter uma instrução de repetição `while`.

**4.9** Identifique e corrija os erros em cada uma das seguintes:

- a) `while ( c <= 5 )`  
`{`  
`product *= c;`  
`c++;`  
b) `cin << value;`  
c) `if ( gender == 1 )`  
`cout << "Woman" << endl;`  
`else;`  
`cout << "Man" << endl;`

**4.10** O que há de errado com a instrução de repetição `while` a seguir?

```
while (z >= 0)
 sum += z;
```

## Respostas dos exercícios de revisão

**4.1** a) Sequência, seleção e repetição. b) `if...else`. c) Controlada por contador ou definida. d) Sentinela, sinal, flag ou dummy.

**4.2** `x = x + 1;`  
`x += 1;`  
`++x;`  
`x++;`

**4.3** a) `z = x++ + y;`  
b) `if ( count > 10 )`  
`cout << "Count is greater than 10" << endl;`  
c) `total -= --x;`  
d) `q %= divisor;`  
`q = q % divisor;`

**4.4** a) `int sum;`  
`int x;`  
b) `x = 1;`  
c) `sum = 0;`  
d) `sum += x;`  
ou  
`sum = sum + x;`  
e) `cout << "The sum is: " << sum << endl;`

**4.5** Examine o código a seguir:

```
1 // Exercício 4.5 Solução: ex04_05.cpp
2 // Calcula a soma dos inteiros de 1 a 10.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9 int sum; // armazena a soma de inteiros de 1 a 10
10 int x; // contador
11
12 x = 1; // contagem de 1
13 sum = 0; // inicializa a soma
14
15 while (x <= 10) // faz o loop 10 vezes
16 {
17 sum += x; // adiciona x à soma
```

```

18 x++; // incrementa x
19 } // fim do while
20
21 cout << "The sum is: " << sum << endl;
22 return 0; // indica terminação bem-sucedida
23 } // fim de main

```

The sum is: 55

- 4.6** a) product = 25, x = 6;  
b) quotient = 0, x = 6;

```

1 // Exercício 4.6 Solução: ex04_06.cpp
2 // Calcula o valor de produto e quociente.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9 int x = 5;
10 int product = 5;
11 int quotient = 5;
12
13 // parte a
14 product *= x++; // instrução da parte a
15 cout << "Value of product after calculation: " << product << endl;
16 cout << "Value of x after calculation: " << x << endl << endl;
17
18 // parte b
19 x = 5; // reinicializa o valor de x
20 quotient /= ++x; // instrução da parte b
21 cout << "Value of quotient after calculation: " << quotient << endl;
22 cout << "Value of x after calculation: " << x << endl << endl;
23 return 0; // indica terminação bem-sucedida
24 } // fim de main

```

Value of product after calculation: 25  
Value of x after calculation: 6

Value of quotient after calculation: 0  
Value of x after calculation: 6

- 4.7** a) cin >> x;  
b) cin >> y;  
c) i = 1;  
d) power = 1;  
e) power \*= x;  
ou  
power = power \* x;  
f) i++;  
g) if ( i <= y )  
h) cout << power << endl;

- 4.8** Examine o código a seguir:



```

1 // Exercício 4.8 Solução: ex04_08.cpp
2 // Eleva x à potência de y.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 int main()
9 {
10 int x; // base
11 int y; // expoente
12 int i; // conta de 1 a y
13 int power; // utilizado para calcular x elevado à potência de y
14
15 i = 1; // inicializa i para começar a contar de 1
16 power = 1; // inicializa power
17
18 cout << "Enter base as an integer: "; // solicita a base
19 cin >> x; // insere a base
20
21 cout << "Enter exponent as an integer: "; // solicita expoente
22 cin >> y; // insere o expoente
23
24 // conta de 1 a y e multiplica potência por x a cada vez
25 while (i <= y)
26 {
27 power *= x;
28 i++;
29 } // fim do while
30
31 cout << power << endl; // exibe o resultado
32 return 0; // indica terminação bem-sucedida
33 } // fim de main

```

```

Enter base as an integer: 2
Enter exponent as an integer: 3
8

```

- 4.9** a) Erro: Está faltando a chave direita de fechamento do corpo do `while`.  
Correção: Adicionar a chave direita de fechamento depois da instrução `c++;`.
- b) Erro: Utilizou inserção de fluxo em vez de extração de fluxo.  
Correção: Alterar `<<` para `>>`.
- c) Erro: Ponto-e-vírgula depois de `else` resulta em um erro de lógica. A segunda instrução de saída sempre será executada.  
Correção: Remover ponto-e-vírgula depois de `else`.
- 4.10** O valor da variável `z` nunca é alterado na instrução `while`. Portanto, se a condição de continuação do loop (`z >= 0`) for inicialmente `true`, um loop infinito é criado. Para evitar o loop infinito, `z` deve ser decrementado de modo que acabe se tornando menor que 0.

## Exercícios

**4.11** Identifique e corrija o(s) erro(s) em cada um dos seguintes:

- a) `if ( age >= 65 );`  
`cout << "Age is greater than or equal to 65" << endl;`  
`else`  
`cout << "Age is less than 65 << endl";`
- b) `if ( age >= 65 )`  
`cout << "Age is greater than or equal to 65" << endl;`  
`else;`  
`cout << "Age is less than 65 << endl";`

```

c) int x = 1, total;

 while (x <= 10)
 {
 total += x;
 x++;
 }
d) While (x <= 100)
 total += x;
 x++;
e) while (y > 0)
 {
 cout << y << endl;
 y++;
 }

```

#### 4.12 O que o programa a seguir imprime?

```

1 // Exercício 4.12: ex04_12.cpp
2 // O que esse programa imprime?
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9 int y; // declara y
10 int x = 1; // inicializa x
11 int total = 0; // inicializa o total
12
13 while (x <= 10) // faz o loop 10 vezes
14 {
15 y = x * x; // realiza os cálculos
16 cout << y << endl; // gera a saída dos resultados
17 total += y; // adiciona y a total
18 x++; // incrementa o contador x
19 } // fim do while
20
21 cout << "Total is " << total << endl; // exibe o resultado
22 return 0; // indica terminação bem-sucedida
23 } // fim de main

```

Para os exercícios 4.13 a 4.16, siga cada um destes passos:

- Leia a declaração do problema.
- Formule o algoritmo utilizando pseudocódigo e refinamento passo a passo de cima para baixo.
- Escreva um programa C++.
- Teste, depure e execute o programa C++.

**4.13** Os motoristas se preocupam com o consumo de combustível dos seus automóveis. Um motorista monitorou vários tanques cheios de gasolina registrando a quilometragem dirigida e a quantidade de combustível em litros utilizados para cada tanque cheio. Desenvolva um programa C++ que utiliza uma instrução `while` e para inserir os quilômetros percorridos e a quantidade de litros de gasolina utilizados para cada taque. O programa deve calcular e exibir o consumo em quilômetros/litro para cada tanque cheio e imprimir a quilometragem combinada e a soma total de litros de combustível consumidos até esse ponto.

```

Entre com a quilometragem (-1 para sair): 287
Entre com os litros: 13
km/litro deste tanque: 22.076923
Total km/litro: 22.076923

```

```

Entre com a quilometragem (-1 para sair): 200
Entre com os litros: 10
km/litro deste tanque: 20.000000
Total km/litro: 21.173913

```

```

Entre com a quilometragem (-1 para sair): 120
Entre com os litros: 5
km/litro deste tanque: 24.000000
Total km/litro: 21.678571

```

```

Entre com a quilometragem: (-1 para sair): -1

```

**4.14** Desenvolva um programa C++ que determinará se um cliente de uma loja de departamentos excedeu o limite de crédito em uma conta corrente. Para cada cliente, os seguintes fatos estão disponíveis:

- Número de conta (um inteiro)
- Balanço no início do mês
- Total de todos os itens cobrados desse cliente no mês
- Total de pagamentos feitos pelo cliente no mês
- Limite autorizado de crédito

O programa deve utilizar uma instrução `while` para inserir cada um desses fatos, calcular o novo saldo (= saldo inicial + taxas – créditos) e determinar se o novo saldo excede o limite de crédito do cliente. Para aqueles clientes cujo limite de crédito é excedido, o programa deve exibir o número da conta do cliente, o limite de crédito, o novo saldo e a mensagem 'Limite de crédito excedido'.

```

Entre com o numero da conta (-1 para terminar): 100
Entre com o saldo inicial: 5394.78
Entre com o total de taxas: 1000.00
Entre com o total de creditos: 500.00
Entre com o limite de credito: 5500.00
Novo saldo: 5894.78
Conta: 100
Limite de credito: 5500.00
Saldo: 5894.78
Limite de credito ultrapassado.

```

```

Entre com o numero da conta (ou -1 para sair): 200
Entre com o saldo inicial: 1000.00
Entre com o total de taxas: 123.45
Entre com o total de creditos: 321.00
Entre com o limite de credito: 1500.00
Novo saldo: 802.45

```

```

Entre com o numero da conta (ou -1 sair): 300
Entre com o saldo inicial: 500.00
Entre com o total de taxas: 274.73
Entre com o total de creditos: 100.00
Entre com o limite de credito: 800.00
Novo saldo: 674.73

```

```

Entre com o numero da conta (ou -1 sair): -1

```

**4.15** Uma grande indústria química paga sua equipe de vendas por comissão. Os vendedores recebem \$ 200 por semana mais 9% de suas vendas brutas por semana. Por exemplo, um vendedor que comercializa um valor de \$ 5.000 em produtos químicos por semana recebe \$ 200 mais

9% de \$ 5.000, ou um total de \$ 650. Desenvolva um programa em C++ que utiliza uma instrução `while` para inserir as vendas brutas de cada vendedor durante a última semana e calcula e exibe os rendimentos desse vendedor. Processe os números de um vendedor por vez.

Entre com as vendas em dolar (-1 para terminar): **5000.00**  
Salario: \$650.00

Entre com as vendas em dolar (-1 para terminar): **6000.00**  
Salario: \$740.00

Entre com as vendas em dolar (-1 para terminar): **7000.00**  
Salario: \$830.00

Entre com as vendas em dolar (-1 para terminar): **-1**

- 4.16** Desenvolva um programa em C++ que utiliza uma instrução `while` para determinar o pagamento bruto de cada um dos vários funcionários. A empresa paga 'hora normal' pelas primeiras 40 horas trabalhadas por empregado e paga 'horas extras' com 50% de gratificação para todas as horas trabalhadas além das primeiras 40 horas. Você recebe uma lista dos empregados da empresa, o número de horas trabalhadas por empregado na última semana e o salário-hora de cada empregado. Seu programa deve aceitar a entrada dessas informações para cada empregado e então determinar e exibir o salário bruto do empregado.

Entre com as horas trabalhadas (-1 para terminar): **39**  
Entre com o valor por hora trabalhada (\$00.00): **10.00**  
Salario: \$390.00

Entre com as horas trabalhadas (-1 para terminar): **40**  
Entre com o valor por hora trabalhada (\$00.00): **10.00**  
Salario: \$400.00

Entre com as horas trabalhadas (-1 para terminar): **41**  
Entre com o valor por hora trabalhada (\$00.00): **10.00**  
Salario: \$415.00

Entre com as horas trabalhadas (-1 para terminar): **-1**

- 4.17** O processo de localizar o maior número (isto é, o máximo de um grupo de valores) é freqüentemente utilizado em aplicativos de computador. Por exemplo, um programa que determina o vencedor de uma competição de vendas insere o número de unidades vendidas por vendedor. O vendedor que vende mais unidades ganha a competição. Escreva um programa em pseudocódigo, e então um programa em C++, que utiliza uma instrução `while` para determinar e imprimir o maior número dos dez números inseridos pelo usuário. Seu programa deve utilizar três variáveis, como segue:

**counter:** Um contador para contar até 10 (isto é, monitorar quantos números foram inseridos e determinar quando todos os 10 números foram processados).  
**number:** A entrada numérica atual para o programa.  
**largest:** O maior número encontrado até agora.

- 4.18** Escreva um programa em C++ que utiliza uma instrução `while` e a sequência de escape de tabulação, `\t`, para imprimir a seguinte tabela de valores:

| N | 10*N | 100*N | 1000*N |
|---|------|-------|--------|
| 1 | 10   | 100   | 1000   |
| 2 | 20   | 200   | 2000   |
| 3 | 30   | 300   | 3000   |
| 4 | 40   | 400   | 4000   |
| 5 | 50   | 500   | 5000   |

- 4.19** Utilizando uma abordagem semelhante àquela do Exercício 4.17, identifique os *dois* maiores valores entre os dez números. [Nota: Cada número deve ser inserido apenas uma vez.]

**4.20** O programa de resultados de teste da Figura 4.16–Figura 4.18 supõe que qualquer valor inserido pelo usuário que não for 1 deve ser 2. Modifique o aplicativo para validar suas entradas. Em qualquer entrada, se o valor entrado for diferente de 1 ou 2, continua o loop até o usuário inserir um valor correto.

**4.21** O que o programa a seguir imprime?

```

1 // Exercício 4.21: ex04_21.cpp
2 // O que esse programa imprime?
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9 int count = 1; // inicializa a contagem
10
11 while (count <= 10) // faz o loop 10 vezes
12 {
13 // saída da linha de texto
14 cout << (count % 2 ? "****" : "++++++") << endl;
15 count++; // incrementa a contagem
16 } // fim do while
17
18 return 0; // indica terminação bem-sucedida
19 } // fim de main

```

**4.22** O que o programa a seguir imprime?

```

1 // Exercício 4.22: ex04_22.cpp
2 // O que esse programa imprime?
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9 int row = 10; // inicializa a linha
10 int column; // declara a coluna
11
12 while (row >= 1) // faz loop até linha < 1
13 {
14 column = 1; // configura coluna como 1 quando a iteração inicia
15
16 while (column <= 10) // faz o loop 10 vezes
17 {
18 cout << (row % 2 ? "<" : ">"); // saída
19 column++; // incrementa coluna
20 } // fim do while interno
21
22 row--; // decrementa linha
23 cout << endl; // inicia nova linha de saída
24 } // fim do while externo
25
26 return 0; // indica terminação bem-sucedida
27 } // fim de main

```

**4.23** (*Problema do else oscilante*) Mostre a saída de cada um dos seguintes quando  $x$  é 9 e  $y$  é 11 e quando  $x$  é 11 e  $y$  é 9. Observe que o compilador ignora o recuo em um programa C++. O compilador C++ sempre associa um `else` com o `if` anterior a menos que ele seja

instruído a fazer de modo diferente pela colocação de chaves `{}`. À primeira vista, o programador pode não estar certo de qual `if` corresponde a qual `else`, o que é referido como problema do ‘`else` oscilante’ (‘`dangling-else`’). Eliminamos o recuo do código a seguir para tornar o problema mais desafiador. [Dica: Aplique as convenções de recuo que você aprendeu.]

```
a) if (x < 10)
 if (y > 10)
 cout << "*****" << endl;
 else
 cout << "#####" << endl;
 cout << "$$$$$" << endl;
b) if (x < 10)
 {
 if (y > 10)
 cout << "*****" << endl;
 }
 else
 {
 cout << "#####" << endl;
 cout << "$$$$$" << endl;
 }
```

**4.24** (*Outro problema de `else` oscilante*) Modifique o seguinte código para produzir a saída mostrada. Utilize técnicas de recuo adequadas. Você não deve fazer nenhuma alteração além de inserir chaves. O compilador ignora o recuo em um programa C++. Eliminamos o recuo do seguinte código para tornar o problema mais desafiador. [Nota: É possível que não seja necessária nenhuma modificação.]

```
if (y == 8)
if (x == 5)
 cout << "@@@@@" << endl;
else
 cout << "#####" << endl;
 cout << "$$$$$" << endl;
 cout << "&&&&&" << endl;
```

a) Supondo que  $x = 5$  e  $y = 8$ , a seguinte saída é produzida.

```
@@@@@
$$$$$
&&&&&
```

b) Supondo que  $x = 5$  e  $y = 8$ , a seguinte saída é produzida.

```
@@@@@
```

c) Supondo que  $x = 5$  e  $y = 8$ , a seguinte saída é produzida.

```
@@@@@
&&&&&
```

d) Supondo que  $x = 5$  e  $y = 7$ , a seguinte saída é produzida. [Nota: Todas as três últimas instruções de saída depois do `else` são partes de um bloco.]

```
####
$$$$$
&&&&&
```

**4.25** Escreva um programa que lê o tamanho do lado de um quadrado e, então, imprime um quadrado vazado com asteriscos e espaços em branco. Seu programa deve trabalhar com quadrados de todos os tamanhos entre 1 e 20. Por exemplo, se seu programa lê um tamanho de 5, ele deve imprimir



```

* *
* *
* *
* *

```

**4.26** Um palíndromo é um número ou uma frase de texto que é lido da mesma forma da esquerda para a direita e da direita para a esquerda. Por exemplo, cada um dos seguintes inteiros de cinco dígitos é um palíndromo: 12321, 55555, 45554 e 11611. Escreva um programa que lê em um inteiro de cinco dígitos e determine se ele é ou não um palíndromo. [Dica: Utilize os operadores de divisão e módulo para separar o número em seus dígitos individuais.]

**4.27** Insira um inteiro contendo somente 0s e 1s (isto é, um inteiro 'binário') e imprima seu equivalente decimal. Utilize os operadores de módulo e divisão para pegar os dígitos do número 'binário' um de cada vez da direita para a esquerda. De modo muito semelhante ao sistema de números decimais, em que o dígito mais à direita tem um valor posicional de 1, o próximo dígito à esquerda tem um valor posicional de 10, depois 100, depois 1.000 e assim por diante, no sistema de números binários o dígito mais à direita tem um valor posicional de 1, o próximo dígito à esquerda tem um valor posicional de 2, depois 4, depois 8 e assim por diante. Assim, o número decimal 234 pode ser interpretado como  $2 * 100 + 3 * 10 + 4 * 1$ . O equivalente decimal do binário 1101 é  $1 * 1 + 0 * 2 + 1 * 4 + 1 * 8$  ou  $1 + 0 + 4 + 8$ , ou 13. [Nota: Recomendamos que leitor não familiarizado com números binários consulte o Apêndice D.]

**4.28** Escreva um programa que exhibe o padrão de tabuleiro mostrado a seguir. Seu programa deve utilizar apenas três instruções de saída, uma de cada das seguintes formas:

```
cout << " * ";
cout << " ";
cout << endl;
```

```
* * * * *
 * * * * *
* * * * *
* * * * *
 * * * * *
* * * * *
 * * * * *
* * * * *
```

**4.29** Escreva um programa que imprime as potências do inteiro 2, isto é, 2, 4, 8, 16, 32, 64 etc. O loop `while` não deve terminar (isto é, você deve criar um loop infinito). Para fazer isso, simplesmente utilize a palavra-chave `true` como a expressão para a instrução `while`. O que acontece quando você executa esse programa?

**4.30** Escreva um programa que lê o raio de um círculo (como um valor `double`) e calcula e imprime o diâmetro, a circunferência e a área. Utilize o valor 3,14159 para  $\pi$ .

**4.31** O que há de errado com a seguinte instrução? Forneça a instrução correta para realizar o que o programador provavelmente estava tentando fazer.

```
cout << ++(x + y);
```

**4.32** Escreva um programa que lê três valores `double` não-zero e determina e imprime se esses valores poderiam representar os lados de um triângulo.

**4.33** Escreva um programa que lê três inteiros não-zero e determina e imprime se eles poderiam ser os lados de um triângulo reto.

**4.34** (Criptografia) Uma empresa quer transmitir dados por telefone, mas está preocupada com a possibilidade de seus telefones estarem grampeados. Todos os dados são transmitidos como inteiros de quatro dígitos. A empresa solicitou que escrevêssemos um programa que encriptasse dados para que pudessem ser transmitidos com maior segurança. Seu programa deve ler um inteiro de quatro dígitos e encriptá-lo como mostrado a seguir: Substitua cada dígito pelo (a soma desse dígito mais 7) módulo 10. Em seguida, troque o primeiro dígito pelo terceiro, o segundo dígito pelo quarto e imprima o inteiro encriptado. Escreva um programa separado que aceita como entrada um inteiro de quatro dígitos criptografado e o descriptografe para formar o número original.

**4.35** O fatorial de um inteiro não negativo  $n$  é escrito como  $n!$  (pronuncia-se 'n fatorial') e é definido como segue:

$$n! = n \cdot (n - 1) \cdot (n - 2) \cdot \dots \cdot 1 \text{ (para valores de } n \text{ maiores que 1)}$$

e

$$n! = 1 \text{ (para } n = 0 \text{ ou } n = 1).$$

Por exemplo,  $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$ , que é 120. Utilize instruções `while` em cada um dos seguintes:

- a) Escreva um programa que lê um inteiro não negativo e calcula e imprime seu fatorial.  
b) Escreva um programa que estima o valor da constante matemática  $e$  utilizando a fórmula:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots$$

Solicite ao usuário a exatidão desejada de  $e$  (isto é, o número de termos na adição).

- c) Escreva um programa que calcula o valor de  $e^x$  utilizando a fórmula

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$$

Solicite ao usuário a exatidão desejada de  $e$  (isto é, o número de termos na adição).

- 4.36** [Nota: Este exercício corresponde à Seção 4.13, uma parte de nosso estudo de caso de engenharia de software.] Descreva em 200 palavras ou menos o que um automóvel é e o que faz. Liste os substantivos e verbos separadamente. No texto, afirmamos que cada substantivo corresponde a um objeto que precisará ser construído para implementar um sistema, nesse caso um carro. Selecione cinco dos objetos que você listou e, para cada um, liste vários atributos e comportamentos. Descreva brevemente como esses objetos interagem entre si e com outros objetos na sua descrição. Você acabou de seguir vários dos passos-chave em um projeto orientado a objetos típico.



*Nem tudo o que pode ser contado importa e nem tudo o que importa pode ser contado.*  
Albert Einstein

*Quem pode controlar seu destino?*  
William Shakespeare

*A chave utilizada é sempre brilhante.*  
Benjamin Franklin

*Inteligência... é a faculdade de criar objetos artificiais, especialmente ferramentas para fazer ferramentas.*  
Henri Bergson

*Toda vantagem no passado é julgada à luz do resultado final.*  
Demóstenes

## Instruções de controle: parte 2

### OBJETIVOS

Neste capítulo, você aprenderá:

- Os princípios básicos da repetição controlada por contador.
- Como utilizar as instruções de repetição `for` e `do...while` para executar instruções em um programa repetidamente.
- A entender a seleção múltipla utilizando a instrução de seleção `switch`.
- Como utilizar as instruções `break` e `continue` para alterar o fluxo de controle.
- Como utilizar os operadores lógicos para formar expressões condicionais complexas em instruções de controle.
- A evitar as consequências de confundir os operadores de igualdade com os operadores de atribuição.

- 5.1 Introdução
- 5.2 Princípios básicos de repetição controlada por contador
- 5.3 A instrução de repetição `for`
- 5.4 Exemplos com a estrutura `for`
- 5.5 Instrução de repetição `do...while`
- 5.6 A estrutura de seleção múltipla `switch`
- 5.7 Instruções `break` e `continue`
- 5.8 Operadores lógicos
- 5.9 Confundindo operadores de igualdade (`==`) com operadores de atribuição (`=`)
- 5.10 Resumo de programação estruturada
- 5.11 Estudo de caso de engenharia de software: identificando estados e atividades dos objetos no sistema ATM (opcional)
- 5.12 Síntese

Resumo | Terminologia | Exercícios de revisão | Respostas dos exercícios de revisão | Exercícios

## 5.1 Introdução

O Capítulo 4 iniciou nossa introdução aos tipos de bloco de construção que estão disponíveis para resolução de problemas. Utilizamos esses blocos de construção para empregar técnicas comprovadas de construção de programa. Neste capítulo, continuamos nossa apresentação da teoria e princípios da programação estruturada introduzindo as instruções de controle restantes do C++. As instruções de controle estudadas neste e no Capítulo 4 nos ajudarão a construir e manipular objetos. Continuamos nossa ênfase inicial em programação orientada a objetos que começou com uma discussão de conceitos básicos no Capítulo 1 e extensos exemplos e exercícios de código orientado a objetos nos capítulos 3–4.

Neste capítulo, demonstramos as instruções `for`, `do...while` e as instruções `switch`. Por uma série de breves exemplos que utilizam `while` e `for`, exploramos os princípios básicos da repetição controlada por contador. Dedicamos uma parte do capítulo à expansão da classe `GradeBook` apresentada nos capítulos 3–4. Em particular, criamos uma versão da classe `GradeBook` que utiliza uma instrução `switch` para contar o número de notas A, B, C, D e F em um conjunto de notas baseadas em letras inseridas pelo usuário. Introduzimos as instruções de controle de programa `break` e `continue`. Discutimos os operadores lógicos, que permitem aos programadores utilizar expressões condicionais mais poderosas em instruções de controle. Examinamos também o erro comum de confundir os operadores de igualdade (`==`) com os operadores de atribuição (`=`) e como evitá-lo. Por fim, resumimos as instruções de controle e as comprovadas técnicas de resolução de problemas do C++ apresentadas neste capítulo e no Capítulo 4.

## 5.2 Princípios básicos de repetição controlada por contador

Esta seção utiliza a instrução de repetição `while` introduzida no Capítulo 4 para formalizar os elementos necessários à realização da repetição controlada por contador. Repetição controlada por contador requer

1. o **nome** de uma variável de controle (ou contador de loop);
2. o **valor inicial** da variável de controle;
3. a **condição de continuação do loop** que testa o **valor final** da variável de controle (isto é, se o loop deve continuar) e
4. o **incremento** (ou **decremento**) pelo qual a variável de controle é modificada a cada passagem pelo loop.

Considere o programa simples na Figura 5.1, que imprime os números de 1 a 10. A declaração na linha 9 *nomeia* a variável de controle (`counter`) e declara essa variável como um inteiro, reserva espaço para ela na memória e a configura como um *valor inicial* de 1. As declarações que requerem inicialização são, de fato, instruções executáveis. Em C++, é mais preciso chamar uma declaração que também reserva memória — como a declaração precedente faz — uma **definição**. Como as definições são também declarações, utilizaremos o termo ‘declaração’ exceto quando a distinção for importante.

A declaração e a inicialização de `counter` (linha 9) também poderiam ter sido realizadas com as instruções

```
int counter; // declara a variável de controle
counter = 1; // inicializa a variável de controle como 1
```

Utilizamos ambos os métodos de inicialização de variáveis.

A linha 14 *incrementa* o contador de loop por 1 toda vez que o corpo do loop é executado. A condição de continuação do loop (linha 11) na instrução `while` determina se o valor da variável de controle é menor que ou igual a 10 (o valor final para o qual a condição é

```

1 // Figura 5.1: fig05_01.cpp
2 // Repetição controlada por contador.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9 int counter = 1; // declara e inicializa a variável de controle
10
11 while (counter <= 10) // condição de continuação do loop
12 {
13 cout << counter << " ";
14 counter++; // incrementa a variável de controle por 1
15 } // fim do while
16
17 cout << endl; // gera a saída de um caractere de nova linha
18 return 0; // terminação bem-sucedida
19 } // fim de main

```

1 2 3 4 5 6 7 8 9 10

**Figura 5.1** Repetição controlada por contador.

true). Observe que o corpo deste `while` executa mesmo quando a variável de controle é 10. O loop termina quando a variável de controle é maior que 10 (isto é, quando `counter` se torna 11).

A Figura 5.1 pode se tornar mais concisa inicializando `counter` como 0 e substituindo a instrução `while` por

```

while (++counter <= 10) // condição de continuação do loop
 cout << counter << " ";

```

Esse código salva uma instrução, porque a incrementação é feita diretamente na condição `while` antes de a condição ser testada. Além disso, o código elimina as chaves em torno do corpo de `while`, porque `while` agora contém somente uma instrução. Codificar de modo tão condensado exige certa prática e pode resultar em programas mais difíceis de ler, depurar, modificar e manter.



### Erro comum de programação 5.1

*Os valores de ponto flutuante são aproximados, portanto controlar a contagem de loops com variáveis de ponto flutuante pode resultar em valores de contador imprecisos e testes imprecisos para terminação.*



### Dica de prevenção de erro 5.1

*Controle a contagem do loop com valores de inteiro.*



### Boa prática de programação 5.1

*Coloque uma linha em branco antes e depois de cada instrução de controle para destacá-la no programa.*



### Boa prática de programação 5.2

*Muitos níveis de aninhamento podem tornar um programa difícil de entender. De modo geral, tente evitar o uso de mais de três níveis de recuo.*



### Boa prática de programação 5.3

*O espaçamento vertical acima e abaixo de instruções de controle e o recuo do corpo das instruções de controle dentro dos cabeçalhos de instrução de controle fornecem aos programas uma aparência bidimensional que melhora significativamente a legibilidade.*

### 5.3 A instrução de repetição for

A Seção 5.2 apresentou os princípios básicos da repetição controlada por contador. A instrução `while` pode ser utilizada para implementar qualquer loop controlado por contador. O C++ também fornece a **instrução de repetição `for`**, que especifica os detalhes da repetição controlada por contador em uma única linha de código. Para ilustrar a capacidade de `for`, vamos reescrever o programa da Figura 5.1. O resultado é mostrado na Figura 5.2.

Quando a instrução `for` (linhas 11–12) começa a executar, a variável de controle `counter` é declarada e inicializada como 1. Então, a condição de continuação do loop `counter <= 10` é verificada. O valor inicial de `counter` é 1, então a condição é satisfeita e a instrução do corpo (linha 12) imprime o valor de `counter`, isto é, 1. Em seguida, a expressão `counter++` incrementa a variável de controle `counter` e o loop inicia novamente com o teste de continuação do loop. A variável de controle é agora igual a 2; portanto, o valor final não é excedido e o programa realiza a instrução de corpo novamente. Esse processo continua até o corpo do loop ter executado 10 vezes e a variável de controle `counter` ser incrementada para 11 — isso faz com que o teste de continuação do loop (linha 11 entre os pontos-e-vírgulas) falhe e com que a repetição termine. O programa continua executando a primeira instrução depois da instrução `for` (nesse caso, a instrução de saída na linha 14).

#### Componentes do cabeçalho da instrução `for`

A Figura 5.3 oferece um exame mais minucioso do cabeçalho da instrução `for` (linha 11) da Figura 5.2. Observe que o cabeçalho da instrução `for` ‘faz tudo’ — ele especifica cada um dos itens necessários para repetição controlada por contador com uma variável de controle. Se houver mais de uma instrução no corpo de `for`, as chaves são necessárias para incluir o corpo do loop.

Note que a Figura 5.2 utiliza a condição de continuação do loop `counter <= 10`. Se o programador escrevesse `counter < 10` incorretamente, então o loop executaria apenas 9 vezes. Esse é um erro *off-by-one*.

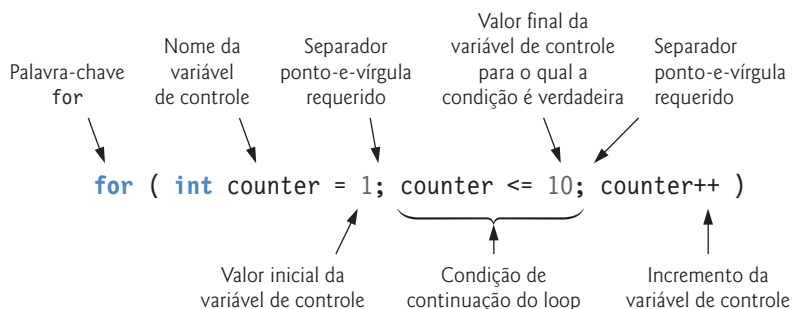
```

1 // Figura 5.2: fig05_02.cpp
2 // Repetição controlada por contador com a instrução for.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9 // cabeçalho da instrução for inclui inicialização,
10 // condição de continuação do loop e incremento.
11 for (int counter = 1; counter <= 10; counter++)
12 cout << counter << " ";
13
14 cout << endl; // gera a saída de um caractere de nova linha
15 return 0; // indica terminação bem-sucedida
16 } // fim de main

```

1 2 3 4 5 6 7 8 9 10

**Figura 5.2** Repetição controlada por contador com a instrução `for`.



**Figura 5.3** Componentes de cabeçalho de instrução `for`.





### Erro comum de programação 5.2

Utilizar um operador relacional incorreto ou um valor final incorreto de um contador de loop na condição de uma instrução `while` ou `for` pode causar erros off-by-one.



### Boa prática de programação 5.4

Utilizar o valor final na condição de uma instrução `while` ou `for` e utilizar o operador `<=` relacional ajudará a evitar erros off-by-one. Para um loop utilizado para imprimir os valores de 1 a 10, por exemplo, a condição de continuação do loop deve ser `counter <= 10` em vez de `counter < 10` (que é um erro off-by-one) ou `counter < 11` (que, porém, é correto). Muitos programadores preferem a chamada **contagem baseada em zero**, em que para contar 10 vezes pelo loop, `counter` seria inicializado como zero e o teste de continuação do loop seria `counter < 10`.

A forma geral da instrução `for` é:

```
for (inicialização; condiçãoDeContinuaçãoDoLoop; incremento)
 instrução
```

onde a expressão *inicialização* inicializa a variável de controle do loop, *condiçãoDeContinuaçãoDoLoop* determina se o loop deve continuar executando (essa condição, em geral, contém o valor final da variável de controle pela qual a condição é verdadeira) e *incremento* incrementa a variável de controle. Na maioria dos casos, a instrução `for` pode ser representada por uma instrução `while` equivalente, como segue:

```
inicialização;

while (condiçãoDeContinuaçãoDoLoop)
{
 instrução
 incremento;
}
```

Há uma exceção a essa regra, que discutiremos na Seção 5.7.

Se a expressão *inicialização* no cabeçalho de instrução `for` declara a variável de controle (isto é, o tipo da variável de controle é especificado antes do nome variável), a variável de controle só pode ser utilizada no corpo da instrução `for` — a variável de controle não será conhecida fora da instrução `for`. Essa utilização restrita do nome da variável de controle é conhecida como **escopo** da variável. O escopo de uma variável especifica onde ele pode ser utilizado em um programa. O escopo é discutido em detalhes no Capítulo 6, “Funções e uma introdução à recursão”.



### Erro comum de programação 5.3

Quando a variável de controle de uma instrução `for` é declarada na seção de inicialização do cabeçalho de instrução `for`, utilizar a variável de controle depois do corpo da instrução é um erro de compilação.



### Dica de portabilidade 5.1

No padrão C++, o escopo da variável de controle declarada na seção de inicialização de uma instrução `for` difere do escopo em compiladores C++ mais antigos. Em compiladores pré-padrão, o escopo da variável de controle não termina no fim do bloco que define o corpo da instrução `for`; em vez disso, o escopo termina no fim do bloco que inclui a instrução `for`. O código C++ criado com compiladores C++ pré-padrão pode quebrar quando compilado em compiladores compatíveis com o padrão. Se você estiver trabalhando com compiladores pré-padrão e quiser certificar-se de que seu código funcionará com compiladores compatíveis com o padrão, há duas estratégias de programação defensivas que você pode utilizar: declarar as variáveis de controle com nomes diferentes em cada instrução `for`, ou, se preferir utilizar o mesmo nome para a variável de controle em várias instruções `for`, declarar a variável de controle antes da primeira instrução `for`.

Como veremos, as expressões *inicialização* e *incremento* podem ser listas de expressões separadas por vírgulas. As vírgulas, como utilizadas nessas expressões, são **operadores vírgula**, que garantem que as listas de expressões sejam avaliadas da esquerda para a direita. O operador vírgula tem a precedência mais baixa de todos os operadores C++. O valor e o tipo de uma lista de expressões separada por vírgulas são o valor e o tipo da expressão mais à direita na lista. O operador vírgula é frequentemente utilizado em instruções `for`. Sua aplicação principal é permitir ao programador utilizar múltiplas expressões de inicialização e/ou múltiplas expressões de incremento. Por exemplo, pode haver diversas variáveis de controle em uma única instrução `for` que devem ser inicializadas e incrementadas.



### Boa prática de programação 5.5

*Coloque apenas expressões que envolvem as variáveis de controle nas seções de inicialização e incremento de uma instrução for. As manipulações de outras variáveis devem aparecer antes do loop (se tiverem de executar apenas uma vez, como as instruções de inicialização) ou no corpo do loop (se tiverem de executar uma vez por repetição, como as instruções de incremento ou decremento).*

As três expressões no cabeçalho da instrução for são opcionais (mas os dois separadores ponto-e-vírgula são necessários). Se a condição *DeContinuaçãoDoLoop* for omitida, o C++ pressupõe que a condição é verdadeira, criando assim um loop infinito. Pode-se omitir a expressão *inicialização* se a variável de controle for inicializada anteriormente no programa. Pode-se omitir a expressão *incremento* se o incremento for calculado por instruções no corpo de for ou se o incremento não for necessário. A expressão de incremento na instrução for atua como uma instrução independente no fim do corpo de um for. Portanto, as expressões

```
counter = counter + 1
counter += 1
++counter
counter++
```

são todas equivalentes na parte de incremento da instrução for (quando nenhum outro código aparece aí). Muitos programadores preferem a forma `counter++`, porque os loops for avaliam a expressão de incremento depois que o corpo do loop executa. A forma pós-incremento portanto parece mais natural. A variável sendo incrementada aqui não aparece em uma expressão maior, assim tanto a pré-incrementação como a pós-incrementação realmente têm o mesmo efeito.



### Erro comum de programação 5.4

*Utilizar vírgulas em vez dos dois ponto-e-vírgulas obrigatórios em um cabeçalho for é um erro de sintaxe.*



### Erro comum de programação 5.5

*Colocar um ponto-e-vírgula imediatamente à direita do parêntese direito de um cabeçalho for torna o corpo dessa instrução for uma instrução vazia. Isso normalmente é um erro de lógica.*



### Observação de engenharia de software 5.1

*Colocar ponto-e-vírgula logo depois de um cabeçalho for é às vezes utilizado para criar um loop chamado **loop de retardo**. Esse loop for com um corpo vazio ainda realiza o loop pelo número indicado de vezes, não fazendo nada além de contar. Por exemplo, você poderia utilizar um loop de retardo para tornar lento um programa que está produzindo saídas na tela rápido demais para serem lidas. Mas seja cuidadoso, porque um retardo assim irá variar entre sistemas com diferentes velocidades de processador.*

As expressões de inicialização, condição de continuação do loop e incremento de uma instrução for podem conter expressões aritméticas. Por exemplo, se  $x = 2$  e  $y = 10$ , e  $x$  e  $y$  não são modificados no corpo do loop, o cabeçalho for

```
for (int j = x; j <= 4 * x * y; j += y / x)
```

é equivalente a

```
for (int j = 2; j <= 80; j += 5)
```

O 'incremento' de uma instrução for pode ser negativo, caso em que é realmente um decremento e o loop realmente conta para baixo (como mostrado na Seção 5.4).

Se a condição de continuação de loop é inicialmente falsa, o corpo da instrução for não é realizado. Em vez disso, a execução prossegue com a instrução seguinte ao for.

Freqüentemente, a variável de controle é impressa ou utilizada em cálculos no corpo de uma instrução for, mas isso não é necessário. É comum utilizar a variável de controle para controlar repetição sem nunca mencioná-la no corpo da instrução for.

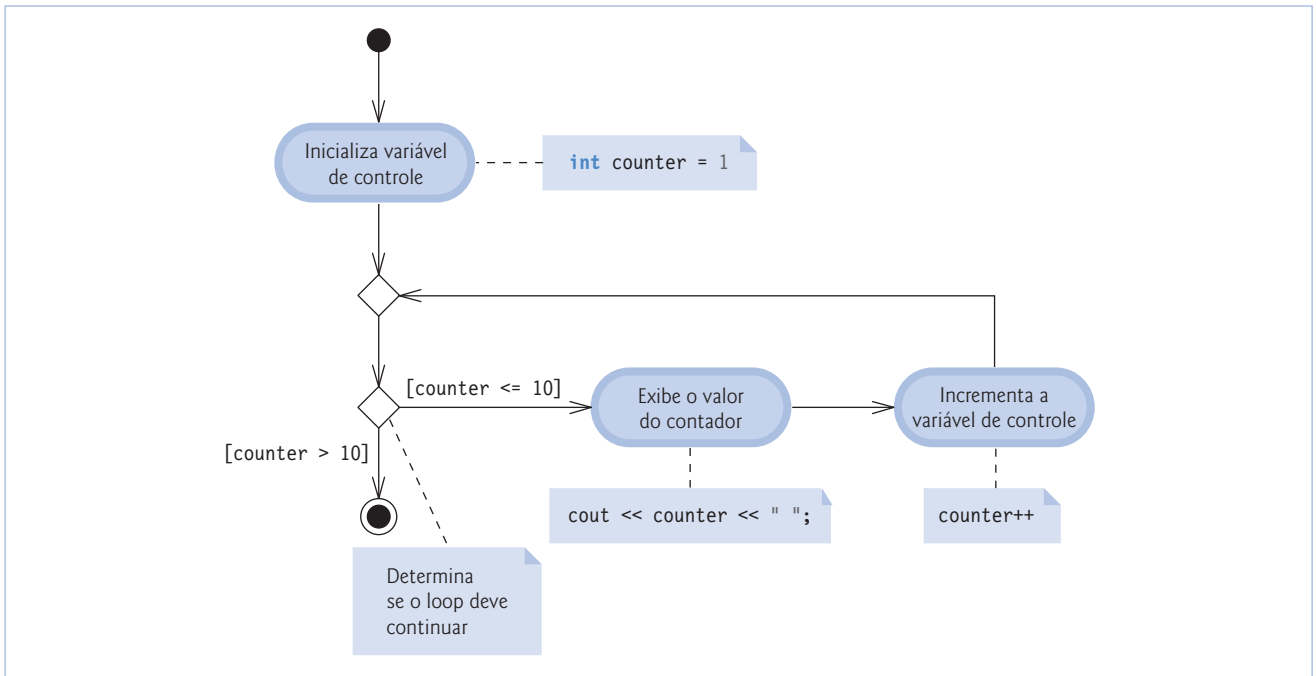


### Dica de prevenção de erro 5.2

*Embora o valor da variável de controle possa ser alterado no corpo de uma instrução for, evite fazer isso porque essa prática pode levar a erros de lógica.*

### Diagrama de atividades UML da instrução for

O diagrama de atividades UML da instrução for é semelhante ao da instrução while (Figura 4.6). A Figura 5.4 mostra o diagrama de atividades da instrução for na Figura 5.2. O diagrama torna claro que a inicialização ocorre uma vez antes de o teste de continuação do loop ser avaliado pela primeira vez, e que o incremento ocorre toda vez por meio do loop *depois* que o corpo da instrução executa. Observe que (além de um estado inicial, setas de transição, uma agregação, um estado final e várias notas) o diagrama contém apenas



**Figura 5.4** Diagrama de atividades UML para a instrução for na Figura 5.2.

estados de ação e uma decisão. Imagine, novamente, que o programador tem um contêiner de diagramas de atividades UML de instruções for vazias — quantas o programador possa precisar para empilhar e aninhar com os diagramas de atividades de outras instruções de controle para formar uma implementação estruturada de um algoritmo. O programador preenche os estados de ação e símbolos de decisão com expressões de ação e condições de guarda apropriadas ao algoritmo.

## 5.4 Exemplos com a estrutura for

Os seguintes exemplos mostram métodos de variação de variável de controle em uma instrução for. Em cada caso, escrevemos o cabeçalho da instrução for apropriado. Observe a alteração no operador relacional para loops que decrementam a variável de controle.

- Altere a variável de controle de 1 a 100 em incrementos de 1.  

```
for (int i = 1; i <= 100; i++)
```
- Altere a variável de controle de 100 para baixo até 1 em incrementos de -1 (isto é, decrementos de 1).  

```
for (int i = 100; i >= 1; i--)
```
- Altere a variável de controle de 7 a 77 em passos de 7.  

```
for (int i = 7; i <= 77; i += 7)
```
- Altere a variável de controle de 20 para baixo até 2 em passos de -2.  

```
for (int i = 20; i >= 2; i -= 2)
```
- Altere a variável de controle sobre a seguinte sequência de valores: 2, 5, 8, 11, 14, 17, 20.  

```
for (int i = 2; i <= 20; i += 3)
```
- Altere a variável de controle sobre a seguinte sequência de valores: 99, 88, 77, 66, 55, 44, 33, 22, 11, 0.  

```
for (int i = 99; i >= 0; i -= 11)
```



### Erro comum de programação 5.6

Não utilizar o operador relacional adequado na condição de continuação do loop de um loop que conta para baixo (como utilizar incorretamente `i <= 1` em vez de `i >= 1` em um loop que conta para baixo até 1) é normalmente um erro de lógica que leva a resultados incorretos quando o programa executa.

*Aplicativo: somando os inteiros pares de 2 a 20*

Os dois próximos exemplos fornecem aplicações simples da instrução for. O programa da Figura 5.5 utiliza uma instrução for para somar os inteiros de 2 a 20. Cada iteração do loop (linhas 12–13) adiciona o valor atual da variável de controle `number` à variável `total`.

Observe que o corpo da instrução `for` na Figura 5.5 não poderia ser realmente fundido na parte de incremento do cabeçalho `for` utilizando o operador vírgula como mostrado a seguir:

```
for (int number = 2; // inicialização
 number <= 20; // condição de continuação do loop
 total += number, number += 2) // total e incremento
 ; // corpo vazio
```



### Boa prática de programação 5.6

Embora as instruções que precedem um `for` e as instruções no corpo de um `for` possam ser freqüentemente fundidas no cabeçalho `for`, fazer isso pode tornar o programa mais difícil de ler, manter, modificar e depurar.



### Boa prática de programação 5.7

Limite o tamanho dos cabeçalhos da instrução de controle a uma única linha, se possível.

#### Aplicativo: cálculos de juros compostos

O próximo exemplo calcula os juros compostos utilizando uma instrução `for`. Considere a seguinte declaração do problema:

*Uma pessoa investe \$ 1.000,00 em uma conta-poupança que rende 5% de juros. Supondo que todos os juros sejam deixados na conta, calcule e imprima o valor em dinheiro na conta ao fim de cada ano durante 10 anos. Utilize a seguinte fórmula para determinar essas quantidades:*

$$a = p(1 + r)^n$$

onde

$p$  é a quantidade original investida (isto é, o principal)

$r$  é a taxa de juros anual

$n$  é o número de anos e

$a$  é a quantidade em depósito no fim do  $n$ -ésimo ano

Esse problema envolve um loop que realiza o cálculo indicado para cada um dos 10 anos que o dinheiro permanece em depósito. A solução é mostrada na Figura 5.6.

A instrução `for` (linhas 28–35) executa o seu corpo 10 vezes, alterando uma variável de controle de 1 a 10 em incrementos de 1. O C++ não inclui um operador de exponenciação, portanto utilizamos a **função de biblioteca-padrão** `pow` (linha 31) para esse propósito. A função `pow( x, y )` calcula o valor de  $x$  elevado à  $y^{\text{ésima}}$  potência. Nesse exemplo, a expressão algébrica  $(1 + r)^n$  é escrita como `pow( 1.0 + rate, year )`, onde a variável `rate` representa  $r$  e a variável `year` representa  $n$ . A função `pow` aceita dois argumentos do tipo `double` e retorna um valor `double`.

```
1 // Figura 5.5: fig05_05.cpp
2 // Somando inteiros com a instrução for.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9 int total = 0; // inicializa o total
10
11 // total de inteiros pares de 2 a 20
12 for (int number = 2; number <= 20; number += 2)
13 total += number;
14
15 cout << "Sum is " << total << endl; // exibe resultados
16 return 0; // terminação bem-sucedida
17 }
```

Sum is 110

**Figura 5.5** Somando inteiros com a instrução `for`.

```

1 // Figura 5.6: fig05_06.cpp
2 // Cálculos de juros compostos com for.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::fixed;
7
8 #include <iomanip>
9 using std::setw; // permite que o programa configure a largura de um campo
10 using std::setprecision;
11
12 #include <cmath> // biblioteca de matemática C++ padrão
13 using std::pow; // permite ao programa utilizar a função pow
14
15 int main()
16 {
17 double amount; // quantia em depósito ao fim de cada ano
18 double principal = 1000.0; // quantia inicial antes dos juros
19 double rate = .05; // taxa de juros
20
21 // exibe cabeçalhos
22 cout << "Year" << setw(21) << "Amount on deposit" << endl;
23
24 // configura o formato de número de ponto flutuante
25 cout << fixed << setprecision(2);
26
27 // calcula quantia de depósito para cada um dos dez anos
28 for (int year = 1; year <= 10; year++)
29 {
30 // calcula nova quantia durante ano especificado
31 amount = principal * pow(1.0 + rate, year);
32
33 // exibe o ano e a quantia
34 cout << setw(4) << year << setw(21) << amount << endl;
35 } // fim do for
36
37 return 0; // indica terminação bem-sucedida
38 } // fim de main

```

| Year | Amount on deposit |
|------|-------------------|
| 1    | 1050.00           |
| 2    | 1102.50           |
| 3    | 1157.63           |
| 4    | 1215.51           |
| 5    | 1276.28           |
| 6    | 1340.10           |
| 7    | 1407.10           |
| 8    | 1477.46           |
| 9    | 1551.33           |
| 10   | 1628.89           |

**Figura 5.6** Cálculos de juros compostos com for.

Esse programa não compilará sem incluir arquivo de cabeçalho `<cmath>` (linha 12). A função `pow` requer dois argumentos `double`. Observe que `year` é um inteiro. O cabeçalho `<cmath>` inclui informações que instruem o compilador a converter o valor de `year` em uma representação `double` temporária antes de chamar a função. Essas informações estão contidas no protótipo da função `pow`. O Capítulo 6 fornece um resumo de outras funções de biblioteca de matemática.



## Erro comum de programação 5.7

*Em geral, esquecer de incluir o arquivo de cabeçalho apropriado ao utilizar funções de biblioteca-padrão (por exemplo, `<cmath>`) em um programa que utiliza funções de biblioteca de matemática) é um erro de compilação.*

*Uma nota de atenção sobre o uso do tipo **double** para valores monetários*

Note que as linhas 17–19 declaram as variáveis `amount`, `principal` e `rate` como tipo `double`. Fizemos isso para simplificar porque estamos lidando com partes fracionárias de valores monetários e precisamos de um tipo que permita pontos decimais em seus valores. Infelizmente, isso pode causar problema. Eis uma explicação simples do que pode dar errado ao utilizar `float` ou `double` para representar quantias monetárias (supondo que `setprecision( 2 )` é utilizado para especificar dois dígitos de precisão na impressão): duas quantias monetárias armazenadas na máquina poderiam ser 14,234 (que é impressa como 14,23) e 18,673 (que é impressa como 18,67). Quando são adicionadas, essas quantidades produzem a soma interna 32,907, que é impressa como 32,91. Assim, sua impressão poderia se parecer com

```
14.23
+ 18.67

32.91
```

mas uma pessoa que adiciona os números individuais como impressos esperaria a soma 32,90! Você foi avisado!



## Boa prática de programação 5.8

*Não utilize variáveis de tipo `float` ou `double` para realizar cálculos monetários. A imprecisão de números de ponto flutuante pode causar erros que resultam em valores monetários incorretos. Nos Exercícios, exploramos o uso de inteiros para realizar cálculos monetários. [Nota: Alguns fornecedores independentes vendem bibliotecas de classes C++ que realizam cálculos monetários precisos. Incluímos vários URLs no Apêndice I.]*

*Utilizando manipuladores de fluxo para formatar a saída numérica*

A instrução de saída na linha 25 antes do loop `for` e a instrução de saída na linha 34 no loop `for` se combinam para imprimir os valores das variáveis `year` e `amount` com a formatação especificada pelos manipuladores de fluxo parametrizado `setprecision` e `setw` e o manipulador de fluxo não parametrizado `fixed`. O manipulador de fluxo `setw( 4 )` especifica que a próxima saída de valor deve aparecer em uma **largura de campo** de 4 — isto é, `cout` imprime o valor com pelo menos 4 posições de caractere. Se o valor a ser enviado para a saída for menor do que a largura de 4 posições de caractere, o valor é **alinhado à direita** no campo por padrão. Se o valor a ser enviado para saída tiver mais de 4 caracteres, a largura de campo é estendida para acomodar o valor inteiro. Para indicar que a saída dos valores deve ser **alinhada à esquerda**, simplesmente gere a saída do manipulador de fluxo não parametrizado `left` (localizado no cabeçalho `<iostream>`). O alinhamento à direita pode ser restaurado gerando a saída do manipulador de fluxo não parametrizado `right`.

A outra formatação nas instruções de saída indica que a variável `amount` é impressa como um valor de ponto fixo com um ponto de fração decimal (especificado na linha 25 com o manipulador de fluxo `fixed`) alinhado à direita em um campo de 21 posições de caractere (especificadas na linha 34 com `setw( 21 )`) e dois dígitos de precisão à direita do ponto de fração decimal (especificado na linha 25 com o manipulador `setprecision( 2 )`). Aplicamos os manipuladores de fluxo `fixed` e `setprecision` ao fluxo de saída (isto é, `cout`) antes do loop `for` porque essas configurações de formato permanecem em vigor até serem alteradas — tais configurações são chamadas **configurações aderentes**. Portanto, elas não precisam ser aplicadas durante cada iteração do loop. Entretanto, a largura de campo especificada com `setw` se aplica somente à próxima saída de valor. Discutimos as poderosas capacidades de formatação de entrada/saída do C++ em detalhes no Capítulo 15.

Observe que o cálculo `1.0 + rate`, que aparece como um argumento para a função `pow`, está contido no corpo da instrução `for`. De fato, esse cálculo produz o mesmo resultado durante cada iteração do loop, então repeti-lo é um desperdício — ele deve ser realizado uma vez antes do loop.



## Dica de desempenho 5.1

*Evite colocar expressões cujos valores não mudam dentro de loops — mas, mesmo se você colocá-las, muitos dos compiladores de otimização sofisticados de hoje irão, automaticamente, colocar essas expressões fora dos loops no código de linguagem de máquina gerado.*



## Dica de desempenho 5.2

*Muitos compiladores contêm recursos de otimização que aprimoram o desempenho do código que você escreve, mas ainda é melhor escrever direito o código desde o início.*

Por diversão, não deixe de experimentar o problema de Peter Minuit no Exercício 5.29. Esse problema demonstra as maravilhas dos juros compostos.



## 5.5 Instrução de repetição do...while

A instrução de repetição do...while é semelhante à instrução while. Na instrução while, o teste de condição de continuação do loop ocorre no começo do loop antes de o corpo do loop executar. A instrução do...while testa a condição de continuação do loop depois de o corpo do loop executar; portanto, o corpo do loop executa sempre pelo menos uma vez. Quando um do...while termina, a execução continua com a instrução depois da cláusula while. Observe que não é necessário utilizar chaves na instrução do...while se houver somente uma instrução no corpo; no entanto, a maioria dos programadores inclui as chaves para evitar confusão entre as instruções while e do...while. Por exemplo,

```
while (condição)
```

normalmente é considerado o cabeçalho de uma instrução while. Um do...while sem chaves em torno do único corpo de instrução aparece como:

```
do
 instrução
while (condição);
```

que pode ser confuso. A última linha — while ( condição ); — poderia ser interpretada erroneamente pelo leitor como uma instrução while contendo como o seu corpo uma instrução vazia. Portanto, do...while com uma única instrução costuma ser escrito como a seguir para evitar confusão:

```
do
{
 instrução
} while (condição);
```



### Boa prática de programação 5.9

*Incluir sempre as chaves em uma instrução do...while ajuda a eliminar a ambigüidade entre a instrução while e a instrução do...while contendo uma instrução.*

A Figura 5.7 utiliza uma instrução do...while para imprimir os números 1–10. No início da instrução do...while, a linha 13 gera saída do valor counter, e a linha 14 incrementa counter. Então o programa avalia o teste de continuação do loop na parte inferior do loop (linha 15). Se a condição for verdadeira, o loop continua a partir da primeira instrução de corpo no do...while (linha 13). Se a condição for falsa, o loop termina e o programa continua com a próxima instrução depois do loop (linha 17).

```
1 // Figura 5.7: fig05_07.cpp
2 // instrução de repetição do...while.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9 int counter = 1; // inicializa o contador
10
11 do
12 {
13 cout << counter << " "; // exibe o contador
14 counter++; // incrementa o contador
15 } while (counter <= 10); // fim da instrução do...while
16
17 cout << endl; // gera a saída de um caractere de nova linha
18 return 0; // indica terminação bem-sucedida
19 } // fim de main
```

1 2 3 4 5 6 7 8 9 10

**Figura 5.7** Instrução de repetição do...while.

### Diagrama de atividades UML da instrução **do...while**

A Figura 5.8 contém o diagrama de atividades UML para a instrução **do...while**. Esse diagrama torna claro que a condição de continuação do loop não é avaliada enquanto o loop não executar os estados de ação do corpo do loop pelo menos uma vez. Compare esse diagrama de atividades com aquele da instrução **while** (Figura 4.6). Novamente, observe que (além de um estado inicial, setas de transição, um agregado, um estado final e várias notas) o diagrama contém apenas estados de ação e uma decisão. Imagine, novamente, que o programador tem acesso a um contêiner de diagramas de atividades UML de instruções **do...while** vazias — quantas o programador possa precisar para empilhar e aninhar com os diagramas de atividades de outras instruções de controle para formar uma implementação estruturada de um algoritmo. O programador preenche os estados de ação e símbolos de decisão com expressões de ação e condições de guarda apropriadas ao algoritmo.

## 5.6 A estrutura de seleção múltipla switch

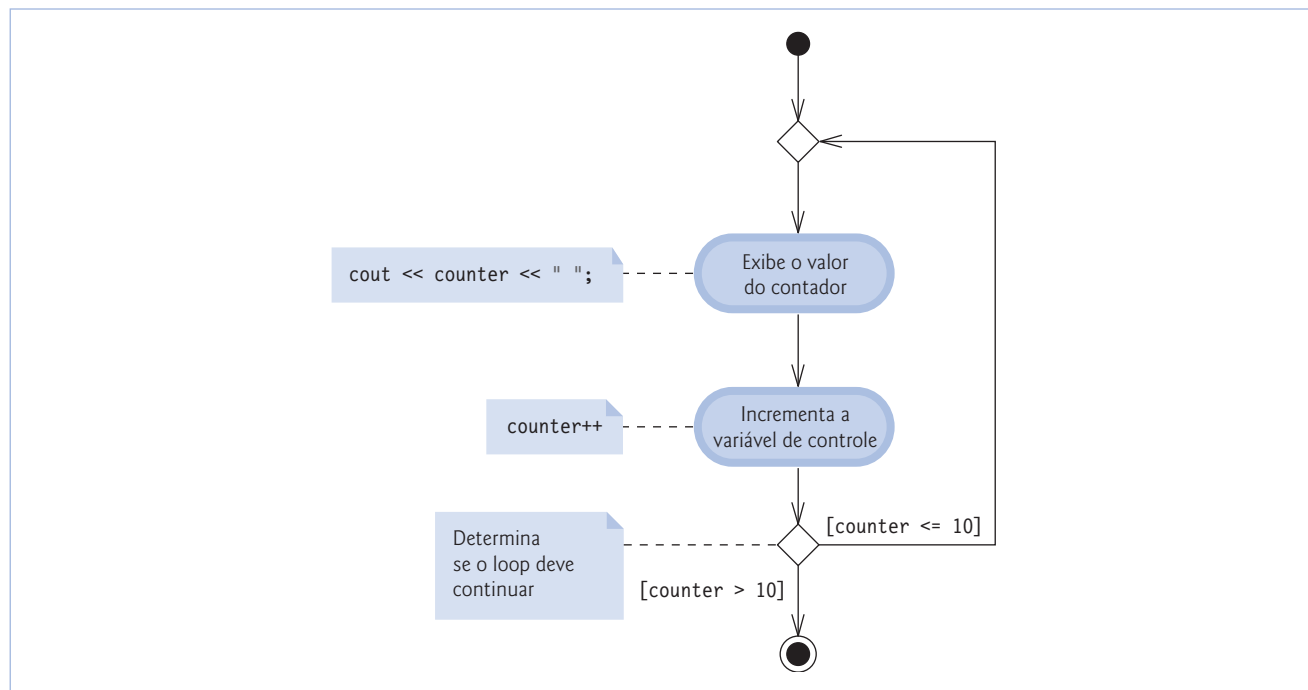
Discutimos a instrução de uma única seleção **if** e a instrução de seleção dupla **if...else** no Capítulo 4. O C++ fornece a instrução de seleção múltipla **switch** para realizar muitas ações diferentes baseadas nos possíveis valores de uma variável ou expressão. Cada ação é associada com o valor de uma **expressão inteira constante** (isto é, qualquer combinação de constantes de caractere e constantes inteiras que são avaliadas como um valor constante inteiro) que a variável ou expressão na qual o **switch** é baseado pode assumir.

### Classe **GradeBook** com a instrução **switch** para contar as notas A, B, C, D e F

No próximo exemplo apresentamos uma versão aprimorada da classe **GradeBook** introduzida no Capítulo 3 e desenvolvida no Capítulo 4. A nova versão da classe solicita que o usuário insira um conjunto de notas baseadas em letras e, então, exibe um resumo do número de alunos que recebeu cada nota. A classe utiliza um **switch** para determinar se cada nota inserida é um A, B, C, D ou F e para incrementar o contador de notas apropriado. A classe **GradeBook** é definida na Figura 5.9 e suas definições de função-membro aparecem na Figura 5.10. A Figura 5.11 mostra entradas e saídas de exemplo do programa **main** que usa a classe **GradeBook** para processar um conjunto de notas.

Como nas versões anteriores da definição de classe, a definição de classe **GradeBook** (Figura 5.9) contém protótipos de função para as funções-membro **setCourseName** (linha 13), **getCourseName** (linha 14) e **displayMessage** (linha 15), bem como para o construtor da classe (linha 12). A definição de classe também declara o membro de dados **private courseName** (linha 19).

A classe **GradeBook** (Figura 5.9) agora contém cinco membros de dados **private** adicionais (linhas 20–24) — variáveis de contador para cada categoria de nota (isto é, A, B, C, D e F). A classe também contém duas funções-membro **public** adicionais — **inputGrades** e **displayGradeReport**. A função-membro **inputGrades** (declarada na linha 16) lê um número arbitrário de notas de letra fornecidas pelo usuário utilizando a repetição controlada por sentinela e atualiza o contador de notas apropriado para cada nota inserida. A função-membro **displayGradeReport** (declarada na linha 17) gera saída de um relatório contendo o número de alunos que recebeu cada nota de letra.



**Figura 5.8** O diagrama de atividades UML para a instrução de repetição **do...while** da Figura 5.7.

```

1 // Figura 5.9: GradeBook.h
2 // Definição da classe GradeBook que conta as notas A, B, C, D e F.
3 // As funções-membro são definidas no GradeBook.cpp
4
5 #include <string> // o programa utiliza classe de string padrão C++
6 using std::string;
7
8 // definição da classe GradeBook
9 class GradeBook
10 {
11 public:
12 GradeBook(string); // o construtor inicializa o nome do curso
13 void setCourseName(string); // função para configurar o nome do curso
14 string getCourseName(); // função para recuperar o nome do curso
15 void displayMessage(); // exibe uma mensagem de boas-vindas
16 void inputGrades(); // insere número arbitrário de notas do usuário
17 void displayGradeReport(); // exibe um relatório baseado nas notas
18 private:
19 string courseName; // nome do curso para esse GradeBook
20 int aCount; // contagem de notas A
21 int bCount; // contagem de notas B
22 int cCount; // contagem de notas C
23 int dCount; // contagem de notas D
24 int fCount; // contagem de notas F
25 }; // fim da classe GradeBook

```

**Figura 5.9** A definição da classe GradeBook.

O arquivo de código-fonte GradeBook.cpp (Figura 5.10) contém as definições de função-membro para a classe GradeBook. Note que as linhas 16–20 no construtor inicializam os cinco contadores de nota como 0 — quando um objeto GradeBook é inicialmente criado, nenhuma nota foi ainda inserida. Como você logo verá, esses contadores são incrementados na função-membro inputGrades quando o usuário inserir as notas. As definições de funções-membro setCourseName, getCourseName e displayMessage são idênticas às encontradas nas versões anteriores da classe GradeBook. Vamos considerar as novas funções-membro GradeBook em detalhes.

```

1 // Figura 5.10: GradeBook.cpp
2 // Definições de função-membro para a classe GradeBook que
3 // utiliza uma instrução switch para contar as notas A, B, C, D e F.
4 #include <iostream>
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include "GradeBook.h" // inclui a definição de classe GradeBook
10
11 // construtor inicializa courseName com string fornecido como argumento;
12 // inicializa membros de dados de contador como 0
13 GradeBook::GradeBook(string name)
14 {
15 setCourseName(name); // valida e armazena courseName
16 aCount = 0; // inicializa a contagem de notas A como 0
17 bCount = 0; // inicializa a contagem de notas B como 0
18 cCount = 0; // inicializa a contagem de notas C como 0
19 dCount = 0; // inicializa a contagem de notas D como 0

```

**Figura 5.10** A classe GradeBook utiliza a instrução switch para contar as notas de letras A, B, C, D e F.

(continua)

```

20 fCount = 0; // inicializa a contagem de notas F como 0
21 } // fim do construtor GradeBook
22
23 // função para configurar o nome do curso; limita o nome a 25 ou menos caracteres
24 void GradeBook::setCourseName(string name)
25 {
26 if (name.length() <= 25) // se o nome tiver 25 ou menos caracteres
27 courseName = name; // armazena o nome do curso no objeto
28 else // se o nome tiver mais que 25 caracteres
29 { // configura courseName como os primeiros 25 caracteres do nome de parâmetro
30 courseName = name.substr(0, 25); // seleciona os primeiros 25 caracteres
31 cout << "Name \"<\" << name << "\" exceeds maximum length (25).\n"
32 << "Limiting courseName to first 25 characters.\n" << endl;
33 } // fim do if...else
34 } // fim da função setCourseName
35
36 // função para recuperar o nome do curso
37 string GradeBook::getCourseName()
38 {
39 return courseName;
40 } // fim da função getCourseName
41
42 // exibe uma mensagem de boas-vindas para o usuário de GradeBook
43 void GradeBook::displayMessage()
44 {
45 // essa instrução chama getCourseName para obter o
46 // nome do curso que esse GradeBook representa
47 cout << "Welcome to the grade book for\n" << getCourseName() << "!\n"
48 << endl;
49 } // fim da função displayMessage
50
51 // insere número arbitrário de notas fornecidas pelo usuário; atualiza o contador de notas
52 void GradeBook::inputGrades()
53 {
54 int grade; // nota inserida pelo usuário
55
56 cout << "Enter the letter grades." << endl
57 << "Enter the EOF character to end input." << endl;
58
59 // faz loop até usuário digitar a sequência de teclas de fim do arquivo
60 while ((grade = cin.get()) != EOF)
61 {
62 // determina que nota foi inserida
63 switch (grade) // instrução switch aninhada em while
64 {
65 case 'A': // a nota era letra A maiúscula
66 case 'a': // ou a minúscula
67 aCount++; // incrementa aCount
68 break; // necessário para fechar switch
69
70 case 'B': // a nota era B maiúscula
71 case 'b': // ou b minúscula
72 bCount++; // incrementa bCount
73 break; // fecha o switch
74
75 case 'C': // a nota era C maiúscula

```

**Figura 5.10** A classe GradeBook utiliza a instrução `switch` para contar as notas de letras A, B, C, D e F.

(continua)

```

76 case 'c': // ou c minúscula
77 cCount++; // incrementa cCount
78 break; // fecha o switch
79
80 case 'D': // a nota era D maiúscula
81 case 'd': // ou d minúscula
82 dCount++; // incrementa dCount
83 break; // fecha o switch
84
85 case 'F': // a nota era F maiúscula
86 case 'f': // ou f minúscula
87 fCount++; // incrementa fCount
88 break; // fecha o switch
89
90 case '\n': // ignora nova linha,
91 case '\t': // tabulações
92 case ' ': // e espaços em entrada
93 break; // fecha o switch
94
95 default: // captura todos os outros caracteres
96 cout << "Incorrect letter grade entered."
97 << " Enter a new grade." << endl;
98 break; // opcional; sairá de switch de qualquer jeito
99 } // fim de switch
100 } // fim do while
101 } // fim da função inputGrades
102
103 // exibe um relatório baseado nas notas inseridas pelo usuário
104 void GradeBook::displayGradeReport()
105 {
106 // gera a saída de resumo de resultados
107 cout << "\n\nNumber of students who received each letter grade:"
108 << "\nA: " << aCount // exibe número de notas A
109 << "\nB: " << bCount // exibe número de notas B
110 << "\nC: " << cCount // exibe número de notas C
111 << "\nD: " << dCount // exibe número de notas D
112 << "\nF: " << fCount // exibe número de notas F
113 << endl;
114 } // fim da função displayGradeReport

```

**Figura 5.10** A classe GradeBook utiliza a instrução `switch` para contar as notas de letras A, B, C, D e F.

(continuação)

### Lendo a entrada de caracteres

O usuário insere notas baseadas em letras para um curso na função-membro `inputGrades` (linhas 52–101). Dentro do cabeçalho `while`, na linha 60, a atribuição entre parênteses (`grade = cin.get()`) executa primeiro. A função `cin.get()` lê um caractere do teclado e armazena esse caractere na variável do tipo inteiro `grade` (declarada na linha 54). Os caracteres são normalmente armazenados em variáveis de tipo `char`; entretanto, os caracteres podem ser armazenados em qualquer tipo de dados inteiro, porque são representados como inteiros de 1 byte no computador. Portanto, você pode tratar um caractere como um inteiro ou como um caractere, dependendo de sua utilização. Por exemplo, a instrução

```

cout << "The character (" << 'a' << ") has the value "
 << static_cast<int> ('a') << endl;

```

imprime o caractere `a` e seu valor inteiro como mostrado a seguir:

```
The character (a) has the value 97
```

O inteiro 97 é a representação numérica do caractere no computador. A maioria dos computadores hoje utiliza o **conjunto de caracteres ASCII (American Standard Code for Information Interchange)**, em que 97 representa a letra minúscula `'a'`. Uma tabela dos caracteres ASCII e seus equivalentes decimais é apresentada no Apêndice B.

As instruções de atribuição têm, como um todo, o valor que é atribuído à variável no lado esquerdo do `=`. Portanto, o valor da expressão de atribuição `grade = cin.get()` tem o mesmo valor que o retornado por `cin.get()` e atribuído à variável `grade`.

O fato de que as instruções de atribuição têm valores pode ser útil para atribuir o mesmo valor a diversas variáveis. Por exemplo,

```
a = b = c = 0;
```

primeiro avalia a atribuição `c = 0` (porque o operador `=` associa da direita para a esquerda). A variável `b` recebe então o valor da atribuição `c = 0` (que é 0). Em seguida, a variável `a` recebe o valor da atribuição `b = (c = 0)` (que também é 0). No programa, o valor da atribuição `grade = cin.get()` é comparado com o valor de EOF (um símbolo cujo acrônimo significa ‘*end-of-file*’, fim do arquivo). Utilizamos EOF (que normalmente tem o valor `-1`) como o valor de sentinela. *Entretanto, você não digita o valor `-1`, nem as letras EOF como o valor de sentinela.* Em vez disso, você digita uma combinação de pressionamentos de teclas dependente do sistema que significa ‘*end-of-file*’ para indicar que não tem mais dados a inserir. EOF é uma constante inteira simbólica definida no arquivo de cabeçalho `<iostream>`. Se o valor atribuído a `grade` for igual a EOF, o loop `while` (linhas 60–100) termina. Escolhemos representar os caracteres inseridos nesse programa como `ints`, porque o EOF tem um valor inteiro.

Em sistemas UNIX/Linux e muitos outros, o fim do arquivo é inserido digitando

```
<ctrl> d
```

em uma linha isolada. Essa notação significa pressionar e manter a tecla *Ctrl* pressionada, e, então, pressionar a tecla *d*. Em outros sistemas, como o Microsoft Windows, o fim do arquivo pode ser inserido digitando

```
<ctrl> z
```

[Nota: Em alguns casos, você deve pressionar *Enter* depois da sequência de tecla precedente. Além disso, os caracteres `^Z`, às vezes, aparecem na tela para representar o fim do arquivo, como é mostrado na Figura 5.11.]



### Dica de portabilidade 5.2

As combinações de teclas pressionadas para inserir o fim do arquivo são dependentes de sistema.



### Dica de portabilidade 5.3

Testar a constante simbólica EOF em vez de `-1` torna os programas mais portáveis. O padrão ANSI/ISO C, a partir do qual o C++ adota a definição de EOF, declara que EOF é um valor inteiro negativo (mas não necessariamente `-1`), então EOF poderia ter diferentes valores em sistemas diferentes.

Nesse programa, o usuário insere notas no teclado. Quando o usuário pressiona a tecla *Enter* (ou *Return*), os caracteres são lidos pela função `cin.get()`, um caractere por vez. Se o caractere inserido não for fim do arquivo, o fluxo de controle entra na instrução `switch` (linhas 63–99), que incrementa o contador de notas de letras apropriado com base na letra inserida.

#### Detalhes da instrução **switch**

A instrução `switch` consiste em uma série de **rótulos case** e um **caso default** opcional. Essas sequências são utilizadas nesse exemplo para determinar qual contador incrementar, com base na nota. Quando o fluxo de controle alcança o `switch`, o programa avalia a expressão entre parênteses (isto é, `grade`) que se segue à palavra-chave `switch` (linha 63). Essa expressão é chamada de **expressão de controle**. A instrução `switch` compara o valor da expressão de controle com cada rótulo `case`. Suponha que o usuário insira a letra `C` como uma nota. O programa compara `C` com cada `case` no `switch`. Se ocorrer uma correspondência (`case 'C':` na linha 75), o programa executa as instruções para esse `case`. Para a letra `C`, a linha 77 incrementa `cCount` por 1. A instrução `break` (linha 78) faz com que o controle de programa prossiga com a primeira instrução depois do `switch` — nesse programa, o controle é transferido para a linha 100. Essa linha marca o fim do corpo do loop `while` que insere notas (linhas 60–100), assim o controle flui para a condição `while` (linha 60) para determinar se o loop deve continuar executando.

Os `cases` em nosso `switch` testam explicitamente as versões em letras minúsculas e maiúsculas das letras `A`, `B`, `C`, `D` e `F`. Observe os `cases` nas linhas 65–66 que testam os valores `'A'` e `'a'` (ambos os quais representam a nota `A`). Listar `cases` consecutivamente dessa maneira, sem instruções entre eles, permite aos `cases` realizar o mesmo conjunto de instruções — quando a expressão de controle é avaliada como `'A'` ou `'a'`, as instruções nas linhas 67–68 serão executadas. Observe que cada `case` pode ter múltiplas instruções. A instrução de seleção `switch` difere de outras instruções de controle porque não exige que as múltiplas instruções em cada `case` estejam entre chaves.

Sem instruções `break`, toda vez que uma correspondência ocorrer no `switch`, as instruções para esse `case` e os `cases` subsequentes executam até uma instrução `break` ou o fim de `switch` ser encontrado. Isso costuma ser referido como ‘*falling through*’, que é o processo em que a instrução percorre sucessivamente os `cases` subsequentes. (Esse recurso é perfeito para escrever um programa conciso que exibe a canção iterativa ‘The Twelve Days of Christmas’ no Exercício 5.28.)



### Erro comum de programação 5.8

Esquecer uma instrução `break` quando esta for necessária em uma instrução `switch` é um erro de lógica.





### Erro comum de programação 5.9

Omitir o espaço entre a palavra `case` e o valor inteiro sendo testado em uma instrução `switch` pode causar um erro de lógica. Por exemplo, escrever `case3:` em vez de `case 3:` simplesmente cria um rótulo não utilizado. Falaremos mais sobre isso no Apêndice E, “Tópicos sobre o código C legado”. Nessa situação, a instrução `switch` não realizará as ações apropriadas quando a expressão de controle de `switch` tiver um valor de 3.

#### Fornecendo um caso **default**

Se não ocorrer nenhuma correspondência entre o valor da expressão controladora e um rótulo `case`, o caso `default` (linhas 95–98) é executado. Utilizamos o caso `default` nesse exemplo para processar todos os valores de expressão de controle que não forem notas válidas nem caracteres de nova linha, tabulações ou espaços (em breve discutiremos como o programa trata esses caracteres de espaço em branco). Se não ocorrer correspondência, o caso `default` executa, e as linhas 96–97 imprimem uma mensagem de erro que indica que uma nota de letra incorreta foi inserida. Se não ocorrer nenhuma correspondência em uma instrução `switch` que não contém um `default`, o controle de programa simplesmente continua com a primeira instrução depois de `switch`.



### Boa prática de programação 5.10

Forneça um caso `default` em instruções `switch`. Os casos não explicitamente testados em uma instrução `switch` sem um caso `default` são ignorados. Incluir um caso `default` concentra o programador na necessidade de processar condições excepcionais. Há situações em que nenhum processamento `default` é necessário. Embora as cláusulas `case` e `default` em uma instrução `switch` possam ocorrer em qualquer ordem, é prática comum colocar a cláusula `default` por último.



### Boa prática de programação 5.11

Em uma instrução `switch` que lista a cláusula `default` por último, a cláusula `default` não requer uma instrução `break`. Alguns programadores incluem esse `break` para clareza e simetria com outros casos.

#### Ignorando caracteres de nova linha, tabulações e em branco na entrada

Observe que as linhas 90–93 na instrução `switch` da Figura 5.10 fazem com que o programa pule os caracteres de nova linha, tabulação e espaço. Ler um caractere por vez pode causar alguns problemas. Para fazer o programa ler os caracteres, devemos enviá-los para o computador pressionando a tecla `Enter` no teclado. Isso coloca um caractere de nova linha na entrada depois do caractere que desejamos processar. Frequentemente, esse caractere de nova linha deve ser especialmente processado para fazer o programa funcionar corretamente. Incluindo os casos precedentes em nossa instrução `switch`, impedimos que a mensagem de erro `default` seja impressa toda vez que uma nova linha, tabulação ou espaço for encontrado na entrada.



### Erro comum de programação 5.10

Não processar a nova linha e outros caracteres de espaço em branco na entrada ao ler caracteres individualmente (um caractere por vez) pode causar erros de lógica.

#### Testando a classe **GradeBook**

A Figura 5.11 cria um objeto `GradeBook` (linha 9). A linha 11 invoca a função-membro `displayMessage` do objeto para realizar saída de uma mensagem de boas-vindas para o usuário. A linha 12 invoca a função-membro `inputGrades` do objeto para ler um conjunto de notas fornecidas pelo usuário e monitora o número de alunos que recebeu cada nota. Observe que a janela de entrada/saída na Figura 5.11 mostra uma mensagem de erro exibida em resposta à inserção de uma nota inválida (isto é, E). A linha 13 invoca a função-membro `GradeBook displayGradeReport` (definida nas linhas 104–114 da Figura 5.10), que gera saída de um relatório baseado nas notas inseridas (como na saída da Figura 5.11).

#### Diagrama de atividades UML da instrução **switch**

A Figura 5.12 mostra o diagrama de atividades UML para a instrução de seleção múltipla `switch` geral. A maioria das instruções `switch` utiliza uma instrução `break` em cada `case` para terminar a instrução `switch` depois de processar o `case`. A Figura 5.12 enfatiza isso incluindo instruções `break` no diagrama de atividades. Sem a instrução `break`, o controle não seria transferido para a primeira instrução após a instrução `switch`, depois que um `case` fosse processado. Em vez disso, o controle seria transferido para as ações do próximo `case`.

O diagrama torna claro que a instrução `break` no fim de um `case` faz com o controle saia imediatamente da instrução `switch`. Novamente, observe que (além de um estado inicial, setas de transição, um estado final e várias notas) o diagrama contém estados de ação e decisões. Além disso, observe que o diagrama utiliza símbolos de agregação para fundir as transições das instruções `break` para o estado final.

```

1 // Figura 5.11: fig05_11.cpp
2 // Cria o objeto GradeBook, insere notas e exibe relatório de notas.
3
4 #include "GradeBook.h" // inclui a definição de classe GradeBook
5
6 int main()
7 {
8 // cria objeto GradeBook
9 GradeBook myGradeBook("CS101 C++ Programming");
10
11 myGradeBook.displayMessage(); // exibe a mensagem de boas-vindas
12 myGradeBook.inputGrades(); // lê as notas fornecidas pelo usuário
13 myGradeBook.displayGradeReport(); // exibe relatório baseado em notas
14 return 0; // indica terminação bem-sucedida
15 } // fim de main

```

```

Welcome to the grade book for
CS101 C++ Programming!

```

```

Enter the letter grades.
Enter the EOF character to end input.

```

```

a
B
c
C
A
d
f
C
E
Incorrect letter grade entered. Enter a new grade.
D
A
b
^Z

```

```

Number of students who received each letter grade:

```

```

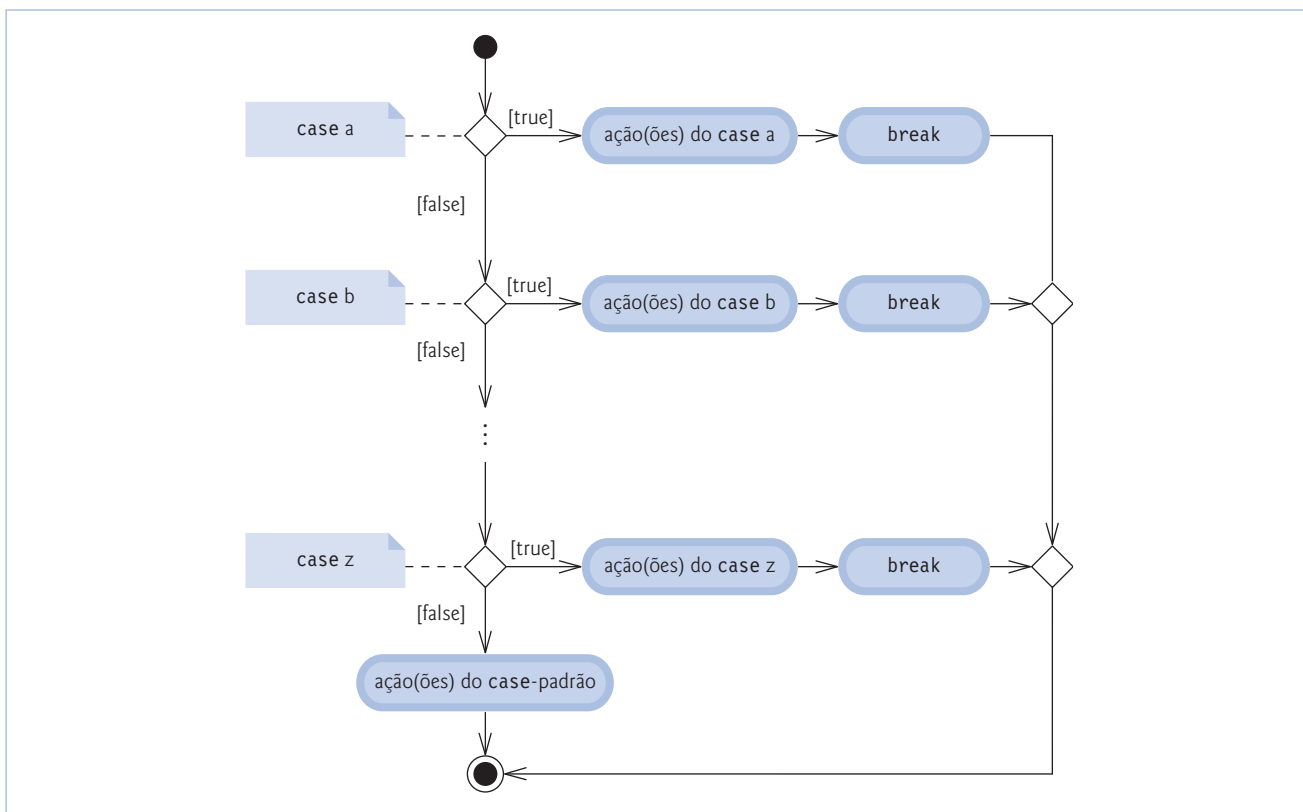
A: 3
B: 2
C: 3
D: 2
F: 1

```

**Figura 5.11** Criando um objeto GradeBook e chamando suas funções-membro.

Imagine, novamente, que o programador tem um contêiner de diagramas de atividades UML de instruções `switch` vazias — quantas o programador possa precisar para empilhar e aninhar com os diagramas de atividades de outras instruções de controle para formar uma implementação estruturada de um algoritmo. O programador preenche os estados de ação e símbolos de decisão com expressões de ação e condições de guarda apropriadas ao algoritmo. Observe que, embora as instruções de controle aninhadas sejam comuns, é raro localizar instruções `switch` aninhadas em um programa.

Ao utilizar a instrução `switch`, lembre-se de que ela só pode ser utilizada para testar uma expressão inteira *constante* — qualquer combinação de constantes de caractere e constantes inteiras que são avaliadas como um valor constante inteiro. Uma constante de caractere é representada como o caractere específico entre aspas simples, como `'A'`. Uma constante de inteiro é simplesmente um valor de inteiro. Além disso, cada rótulo `case` pode especificar apenas uma expressão inteira constante.



**Figura 5.12** Diagrama de atividades UML de instrução de seleção múltipla `switch` com instruções `break`.



### Erro comum de programação 5.11

Especificar uma expressão incluindo variáveis (por exemplo, `a + b`) no rótulo `case` de uma instrução `switch` é um erro de sintaxe.



### Erro comum de programação 5.12

Fornecer rótulos `case` idênticos em uma instrução `switch` é um erro de compilação. Fornecer rótulos `case` contendo expressões diferentes que são avaliadas como o mesmo valor também é um erro de compilação. Por exemplo, colocar `case 4 + 1:` e `case 3 + 2:` na mesma instrução `switch` é um erro de compilação, porque ambos são equivalentes ao `case 5:`.

No Capítulo 13 apresentamos uma maneira mais elegante de implementar a lógica `switch`. Utilizaremos uma técnica chamada polimorfismo para criar programas que são frequentemente mais claros, mais concisos, mais fáceis de manter e mais fáceis de estender que programas que utilizam a lógica `switch`.

#### Notas sobre tipos de dados

O C++ tem tamanhos flexíveis de tipo de dados (ver Apêndice C, “Tipos fundamentais”). Aplicativos diferentes, por exemplo, talvez precisem de inteiros de tamanhos diferentes. O C++ fornece vários tipos de dados para representar inteiros. O intervalo de valores inteiros para cada tipo depende do hardware do computador particular. Além dos tipos `int` e `char`, o C++ fornece os tipos `short` (uma abreviação de `short int`) e `long` (uma abreviação de `long int`). O intervalo mínimo de valores para inteiros `short` é  $-32.768$  a  $32.767$ . Para a vasta maioria de cálculos de inteiro, os inteiros `long` são suficientes. O intervalo mínimo de valores para inteiros `long` é  $-2.147.483.648$  a  $+2.147.483.647$ . Na maioria dos computadores, `ints` são equivalentes a `short` ou `long`. O intervalo de valores de um `int` é pelo menos o mesmo que o de inteiros `short` e não maior que o de inteiros `long`. O tipo de dados `char` pode ser utilizado para representar qualquer dos caracteres no conjunto de caracteres do computador. Também pode ser utilizado para representar inteiros pequenos.



### Dica de portabilidade 5.4

Como `ints` podem variar de tamanho entre sistemas, utilize inteiros `long` se você espera processar inteiros fora do intervalo de  $-32.768$  a  $32.767$  e quiser executar o programa em diversos sistemas de computador diferentes.

**Dica de desempenho 5.3**

*Se a memória for muito cara, talvez seja desejável utilizar tamanhos de inteiro menores.*

**Dica de desempenho 5.4**

*Utilizar tamanhos menores de inteiro pode resultar em um programa mais lento se as instruções de máquina para manipulá-las não forem tão eficientes quanto aquelas para os inteiros de tamanho natural, isto é, inteiros cujo tamanho é igual ao tamanho de palavra da máquina (por exemplo, de 32 bits em uma máquina de 32 bits, de 64 bits em uma máquina de 64 bits). Sempre teste 'atualizações' de eficiência propostas para certificar-se de que elas realmente melhoram o desempenho.*

## 5.7 Instruções break e continue

Além das instruções de seleção e repetição, o C++ fornece instruções `break` e `continue` para alterar o fluxo de controle. A seção precedente mostrou como `break` pode ser utilizado para terminar a execução de uma instrução `switch`. Esta seção discute como utilizar `break` em uma instrução de repetição.

### Instrução `break`

A instrução `break`, quando executada em uma instrução `while`, `for`, `do...while` ou `switch`, ocasiona sua saída imediata dessa instrução. A execução de programa continua com a próxima instrução. Utilizações comuns da instrução `break` são escapar no começo de um loop ou pular o restante de uma instrução `switch` (como na Figura 5.10). A Figura 5.13 demonstra a instrução `break` (linha 14) saindo de uma instrução de repetição `for`.

Quando a instrução `if` detecta que `count` é 5, a instrução `break` executa. Isso termina a instrução `for` e o programa prossegue para a linha 19 (imediatamente depois da instrução `for`), que exibe uma mensagem indicando o valor da variável de controle que termina o loop. A instrução `for` executa completamente o seu corpo por apenas quatro vezes em vez de 10. Observe que a variável de controle `count` é definida fora do cabeçalho de instrução `for`, para que possamos utilizar a variável de controle no corpo do loop e depois que o loop completar sua execução.

```

1 // Figura 5.13: fig05_13.cpp
2 // a instrução break sai de uma instrução for.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9 int count; // variável de controle também utilizada depois que loop termina
10
11 for (count = 1; count <= 10; count++) // itera 10 vezes
12 {
13 if (count == 5)
14 break; // quebra o loop somente se x for 5
15
16 cout << count << " ";
17 } // fim do for
18
19 cout << "\nBroke out of loop at count = " << count << endl;
20 return 0; // indica terminação bem-sucedida
21 } // fim de main

```

```

1 2 3 4
Broke out of loop at count = 5

```

**Figura 5.13** Instrução `break` saindo de uma instrução `for`.

**Instrução *continue***

A instrução *continue*, quando executada em uma instrução *while*, *for* ou *do...while*, ignora as instruções restantes no corpo dessa instrução e prossegue com a próxima iteração do loop. Em instruções *while* e *do...while*, o teste de continuação do loop é imediatamente avaliado depois de a instrução *continue* executar. Na instrução *for*, a expressão de incremento executa e, então, o teste de continuação do loop é avaliado.

A Figura 5.14 utiliza a instrução *continue* (linha 12) em uma instrução *for* para pular a instrução de saída (linha 14) quando o *if* aninhado (linhas 11–12) determina que o valor de *count* é 5. Quando a instrução *continue* executa, o controle de programa continua com o incremento da variável de controle no cabeçalho *for* (linha 9) e faz loop mais cinco vezes.

Na Seção 5.3, declaramos que a instrução *while* e poderia ser utilizada na maioria dos casos para representar a instrução *for*. A exceção ocorre quando a expressão de incremento na instrução *while* segue a instrução *continue*. Nesse caso, o incremento não executa antes de o programa testar a condição de continuação do loop, e o *while* não é executado da mesma maneira que o *for*.

**Boa prática de programação 5.12**

*Alguns programadores consideram que `break` e `continue` violam a programação estruturada. Os efeitos dessas instruções podem ser alcançados por técnicas de programação estruturada que logo aprenderemos, portanto esses programadores não utilizam `break` e `continue`. A maioria dos programadores considera o uso de `break` em instruções `switch` aceitável.*

**Dica de desempenho 5.5**

*As instruções `break` e `continue`, quando utilizadas adequadamente, desempenham mais rapidamente que as técnicas estruturadas correspondentes.*

**Observação de engenharia de software 5.2**

*Há uma tensão entre alcançar engenharia de software de qualidade e alcançar o software de melhor desempenho. Frequentemente, um desses objetivos é alcançado à custa do outro. Para todas as situações, exceto as de desempenho muito alto, aplique a seguinte regra geral: primeiro, faça seu código simples e correto; então, torne-o rápido e pequeno, mas apenas se necessário.*

```

1 // Figura 5.14: fig05_14.cpp
2 // continua instrução que termina uma iteração de uma instrução for.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9 for (int count = 1; count <= 10; count++) // itera 10 vezes
10 {
11 if (count == 5) // se count for 5,
12 continue; // pula o código restante no loop
13
14 cout << count << " ";
15 } // fim do for
16
17 cout << "\nUsed continue to skip printing 5" << endl;
18 return 0; // indica terminação bem-sucedida
19 } // fim de main

```

```

1 2 3 4 6 7 8 9 10
Used continue to skip printing 5

```

**Figura 5.14** A instrução *continue* que termina uma única iteração de uma instrução *for*.

## 5.8 Operadores lógicos

Até agora estudamos somente **condições simples** como `counter <= 10`, `total > 1000` e `number != sentinelValue`. Expressamos essas condições em termos dos operadores relacionais `>`, `<`, `>=` e `<=` e os operadores de igualdade `==` e `!=`. Cada decisão testou precisamente uma condição. Para testar múltiplas condições ao tomar uma decisão, realizamos esses testes em instruções separadas ou em instruções `if` ou `if...else` aninhadas.

O C++ fornece os **operadores lógicos** que são utilizados para formar condições mais complexas combinando condições simples. Os operadores lógicos são `&&` (E lógico), `||` (OU lógico) e `!` (NÃO lógico, também chamado de negação lógica).

### Operador E lógico (&&)

Suponha que quiséssemos assegurar que duas condições fossem *ambas* true antes de escolhermos certo caminho de execução. Nesse caso, podemos utilizar o operador **&&** (E lógico), como segue:

```
if (gender == 1 && age >= 65)
 seniorFemales++;
```

Esta instrução `if` contém duas condições simples. A condição `gender == 1` é utilizada aqui para determinar se uma pessoa é do sexo feminino. A condição `age >= 65` determina se uma pessoa é um(a) cidadão(ã) idoso(a). A condição simples à esquerda do operador `&&` é avaliada primeiro, porque a precedência de `==` é mais alta que a precedência de `&&`. Se necessário, a condição simples à direita do operador `&&` é avaliada em seguida, porque a precedência de `>=` é mais alta que a precedência de `&&`. Como discutiremos em breve, o lado direito de uma expressão E lógica é avaliado somente se o lado esquerdo for true. A instrução `if` então considera a condição combinada

```
gender == 1 && age >= 65
```

Essa condição é true se e unicamente ambas as condições simples forem true. Por fim, se essa condição combinada for de fato true, a instrução no corpo da instrução `if` incrementa a contagem de `seniorFemales`. Se qualquer uma das condições simples for false (ou ambas), então o programa ignora a incrementação e prossegue com a instrução que se segue ao `if`. A condição combinada precedente pode tornar-se mais legível adicionando-se parênteses redundantes:

```
(gender == 1) && (age >= 65)
```



### Erro comum de programação 5.13

*Embora  $3 < x < 7$  seja uma condição matematicamente correta, ela não é avaliada como você talvez imagine em C++. Utilize  $(3 < x \ \&\& \ x < 7)$  para obter a avaliação adequada em C++.*

A Figura 5.15 resume o operador `&&`. A tabela mostra todas as quatro possíveis combinações de valores false e true para *expressão1* e *expressão2*. Essas tabelas costumam ser chamadas de **tabelas-verdade**. O C++ avalia todas as expressões false ou true que incluem operadores relacionais, operadores de igualdade e/ou operadores lógicos.

### Operador OU lógico (||)

Agora vamos considerar o operador `||` (OU lógico). Suponha que quiséssemos assegurar em algum ponto em um programa que qualquer uma *ou* ambas das condições fossem true antes de escolhermos certo caminho de execução. Nesse caso utilizamos o operador `||`, como no segmento de programa seguinte:

```
if ((semesterAverage >= 90) || (finalExam >= 90))
 cout << "Student grade is A" << endl;
```

Essa condição precedente também contém duas condições simples. A condição simples `semesterAverage >= 90` é avaliada para determinar se o aluno merece um 'A' no curso devido a um sólido desempenho ao longo de todo o semestre. A condição simples `finalExam >= 90` é avaliada para determinar se o aluno merece um 'A' no curso devido a um desempenho destacado no exame final. A instrução `if` então considera a condição combinada

```
(semesterAverage >= 90) || (finalExam >= 90)
```

e premia o aluno com um 'A' se qualquer uma ou ambas as condições simples forem true. Observe que a mensagem 'Student grade is A' é impressa a menos que ambas as condições simples sejam false. A Figura 5.16 é uma tabela-verdade para o operador lógico `||`.

O operador `&&` tem uma precedência mais alta que o operador `||`. Ambos os operadores associam-se da esquerda para a direita. Uma expressão contendo os operadores `&&` ou `||` é avaliada somente até que a condição de verdade ou falsidade da expressão seja conhecida. Portanto, a avaliação da expressão

```
(gender == 1) && (age >= 65)
```

pára imediatamente se `gender` não for igual a 1 (isto é, a expressão inteira for false) e continua se `gender` for igual a 1 (isto é, a expressão inteira poderia ainda ser true se a condição `age >= 65` fosse true). Esse recurso de desempenho para a avaliação de expressões com E lógico e OU lógico é chamado **avaliação de curto-circuito**.



| expressão 1 | expressão2 | expressão 1 && expressão2 |
|-------------|------------|---------------------------|
| false       | false      | false                     |
| false       | true       | false                     |
| true        | false      | false                     |
| true        | true       | true                      |

Figura 5.15 Tabela-verdade do operador && (E lógico).

| expressão 1 | expressão2 | expressão 1    expressão2 |
|-------------|------------|---------------------------|
| false       | false      | false                     |
| false       | true       | true                      |
| true        | false      | true                      |
| true        | true       | true                      |

Figura 5.16 Tabela-verdade do operador || (OU lógico).



### Dica de desempenho 5.6

Em expressões que utilizam o operador &&, se as condições separadas forem independentes uma da outra, torne a condição que muito provavelmente deve ser false a condição mais à esquerda. Em expressões utilizando o operador ||, torne a condição que muito provavelmente deve ser true a condição mais à esquerda. Essa utilização de avaliação em curto-circuito pode reduzir o tempo de execução de um programa.

### Operador de negação lógica (!)

O C++ fornece o operador **!** (**NÃO lógico**, também chamado de **negação lógica**) para permitir ao programador ‘inverter’ o significado de uma condição. Diferentemente dos operadores binários && e ||, que combinam duas condições, o operador unário de negação lógica tem apenas uma única condição como um operando. O operador unário de negação lógica é colocado antes de uma condição quando estamos interessados em escolher um caminho de execução se a condição original (sem o operador de negação lógica) for false, como no seguinte segmento de programa:

```
if (!(grade == sentinelValue))
 cout << "The next grade is " << grade << endl;
```

Os parênteses em torno da condição `grade == sentinelValue` são necessários uma vez que o operador de negação lógica tem uma precedência mais alta que o operador de igualdade.

Na maioria dos casos, o programador pode evitar a utilização da negação lógica expressando a condição com um operador relacional ou de igualdade apropriado. Por exemplo, a instrução `if` precedente também pode ser escrita da seguinte maneira:

```
if (grade != sentinelValue)
 cout << "The next grade is " << grade << endl;
```

Essa flexibilidade pode frequentemente ajudar um programador a expressar uma condição de maneira mais ‘natural’ ou conveniente. A Figura 5.17 é uma tabela-verdade para o operador de negação lógica (!).

| expressão | ! expressão |
|-----------|-------------|
| false     | true        |
| true      | false       |

Figura 5.17 Tabela-verdade do operador ! (negação lógica).

*Exemplo de operadores lógicos*

A Figura 5.18 demonstra os operadores lógicos produzindo suas tabelas-verdade. A saída mostra cada expressão que é avaliada e seu resultado bool. Por padrão, valores bool `true` e `false` são exibidos por `cout` e pelo operador de inserção de fluxo como 1 e 0, respectivamente. Entretanto, utilizamos [manipulador de fluxo `boolalpha`](#) na linha 11 para especificar que o valor de cada expressão bool deve ser exibido como a palavra 'true' ou 'false'. Por exemplo, o resultado da expressão `false && false` na linha 12 é `false`, então a segunda linha de saída inclui a palavra 'false'. As linhas 11–15 produzem a tabela-verdade para `&&`. As linhas 18–22 produzem a tabela-verdade para `||`. As linhas 25–27 produzem a tabela-verdade para `!`.

```

1 // Figura 5.18: fig05_18.cpp
2 // Operadores lógicos.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::boolalpha; // faz com que valores bool sejam impressos como "true" ou "false"
7
8 int main()
9 {
10 // cria a tabela-verdade para o operador && (E lógico)
11 cout << boolalpha << "Logical AND (&&)"
12 << "\nfalse && false: " << (false && false)
13 << "\nfalse && true: " << (false && true)
14 << "\ntrue && false: " << (true && false)
15 << "\ntrue && true: " << (true && true) << "\n\n";
16
17 // cria a tabela-verdade para o operador || (OU lógico)
18 cout << "Logical OR (||)"
19 << "\nfalse || false: " << (false || false)
20 << "\nfalse || true: " << (false || true)
21 << "\ntrue || false: " << (true || false)
22 << "\ntrue || true: " << (true || true) << "\n\n";
23
24 // cria a tabela-verdade para o operador ! (negação lógica)
25 cout << "Logical NOT (!)"
26 << "\n!false: " << (!false)
27 << "\n!true: " << (!true) << endl;
28 return 0; // indica terminação bem-sucedida
29 } // fim de main

```

```

Logical AND (&&)
false && false: false
false && true: false
true && false: false
true && true: true

```

```

Logical OR (||)
false || false: false
false || true: true
true || false: true
true || true: true

```

```

Logical NOT (!)
!false: true
!true: false

```

**Figura 5.18** Operadores lógicos.

*Resumo de precedência e associatividade de operadores*

A Figura 5.19 adiciona os operadores lógicos ao gráfico de precedência e associatividade de operadores. Os operadores são mostrados de cima para baixo em ordem decrescente de precedência.

## 5.9 Confundindo operadores de igualdade (==) com operadores de atribuição (=)

Há um tipo de erro que programadores em C++, independentemente de sua experiência, tendem a cometer com tanta frequência que sentimos que ele requer uma seção separada. Esse erro faz acidentalmente a troca dos operadores == (igualdade) por = (atribuição). O que faz com que essas trocas sejam tão prejudiciais é o fato de que elas normalmente não causam erro de sintaxe. Em vez disso, as instruções com esses erros tendem a compilar corretamente e os programas executam até a conclusão, gerando muitas vezes resultados incorretos por meio de erros de lógica em tempo de execução. [Nota: Alguns compiladores publicam um aviso quando = é utilizado em um contexto em que == é normalmente esperado.]

Há dois aspectos de C++ que contribuem para esses problemas. Um é que qualquer expressão que produz um valor pode ser utilizada na parte de decisão de qualquer instrução de controle. Se o valor da expressão é zero, ele é tratado como `false` e, se não for zero, é tratado como `true`. O segundo aspecto é que as atribuições produzem um valor — a saber, o valor atribuído à variável no lado esquerdo do operador de atribuição. Por exemplo, suponha que pretendêssemos escrever

```
if (payCode == 4)
 cout << "You get a bonus!" << endl;
```

mas acidentalmente escrevemos

```
if (payCode = 4)
 cout << "You get a bonus!" << endl;
```

A primeira instrução `if` dá adequadamente um bônus à pessoa cujo `payCode` é igual a 4. A segunda instrução `if` — aquela com o erro — avalia a expressão de atribuição na condição `if` como a constante 4. Qualquer valor diferente de zero é interpretado como `true`, então a condição nessa instrução `if` é sempre `true` e a pessoa sempre recebe um bônus independentemente de qual seja o código de pagamento real! Pior ainda, o código de pagamento foi modificado quando se esperava que ele fosse apenas examinado!



### Erro comum de programação 5.14

Usar o operador == para atribuição e usar o operador = para igualdade são erros de lógica.

| Operadores                                        | Associatividade            | Tipo                 |
|---------------------------------------------------|----------------------------|----------------------|
| ()                                                | da esquerda para a direita | parênteses           |
| ++    -- <code>static_cast&lt; tipo &gt;()</code> | da esquerda para a direita | unário (pós-fixado)  |
| ++    --    +    -    !                           | da direita para a esquerda | unário (prefixado)   |
| *    /    %                                       | da esquerda para a direita | multiplicativo       |
| +    -                                            | da esquerda para a direita | aditivo              |
| <<    >>                                          | da esquerda para a direita | inserção/extração    |
| <    <=    >    >=                                | da esquerda para a direita | relacional           |
| ==    !=                                          | da esquerda para a direita | igualdade            |
| &&                                                | da esquerda para a direita | E lógico             |
|                                                   | da esquerda para a direita | OU lógico            |
| ?:                                                | da direita para a esquerda | ternário condicional |
| =    +=    -=    *=    /=    %=                   | da direita para a esquerda | atribuição           |
| ,                                                 | da esquerda para a direita | vírgula              |

**Figura 5.19** Precedência e associatividade de operadores.



### Dica de prevenção de erro 5.3

Normalmente, programadores escrevem condições como `x == 7` com o nome da variável à esquerda e a constante à direita. Invertendo para que a constante esteja à esquerda e o nome da variável à direita, como em `7 == x`, o programador que acidentalmente substituir o operador `==` por `=` será protegido pelo compilador. O compilador trata isso como um erro de compilação, porque você não pode alterar o valor de uma constante. Isso evitará a potencial devastação de um erro de lógica em tempo de execução.

Diz-se que os nomes de variáveis são *lvalues* (de ‘left values’) porque podem ser utilizados à esquerda do operador de atribuição. Diz-se que as constantes são *rvalues* (para ‘right values’) porque podem ser utilizadas somente à direita do operador de atribuição. Observe que *lvalues* também podem ser utilizados como *rvalues*, mas não vice-versa.

Há outra situação igualmente desagradável. Suponha que o programador queira atribuir um valor a uma variável com uma instrução simples como

```
x = 1;
```

mas, em vez disso, escreve:

```
x == 1;
```

Aqui também esse não é um erro de sintaxe. Em vez disso, o compilador simplesmente avalia a expressão condicional. Se `x` é igual a 1, a condição é `true` e a expressão é avaliada como o valor `true`. Se `x` não é igual a 1, a condição é `false` e a expressão é avaliada como o valor `false`. Independentemente do valor da expressão, não há nenhum operador de atribuição, portanto o valor é simplesmente perdido. O valor de `x` permanece inalterado, causando provavelmente um erro de lógica em tempo de execução. Infelizmente, não temos um truque útil disponível para ajudá-lo com esse problema!



### Dica de prevenção de erro 5.4

Utilize seu editor de textos para procurar todas as ocorrências de `=` no programa e verifique se você tem o operador de atribuição correto ou operador lógico em cada lugar.

## 5.10 Resumo de programação estruturada

Da mesma forma que os arquitetos projetam edifícios empregando o conhecimento coletivo de sua profissão, assim também os programadores devem projetar programas. Nosso campo é mais jovem que a arquitetura e nossa sabedoria coletiva é consideravelmente mais esparsa. Aprendemos que a programação estruturada produz programas mais fáceis de entender, testar, depurar, modificar e até demonstrar como corretos em um sentido matemático do que os programas não estruturados.

A Figura 5.20 usa diagramas de atividades para resumir as instruções de controle do C++. Os estados inicial e final indicam o único ponto de entrada e o único ponto de saída de cada instrução de controle. Conectar símbolos individuais arbitrariamente em um diagrama de atividades pode levar a programas não estruturados. Portanto, a profissão de programador utiliza somente um conjunto limitado de instruções de controle que podem ser combinadas apenas de duas maneiras simples para construir programas estruturados.

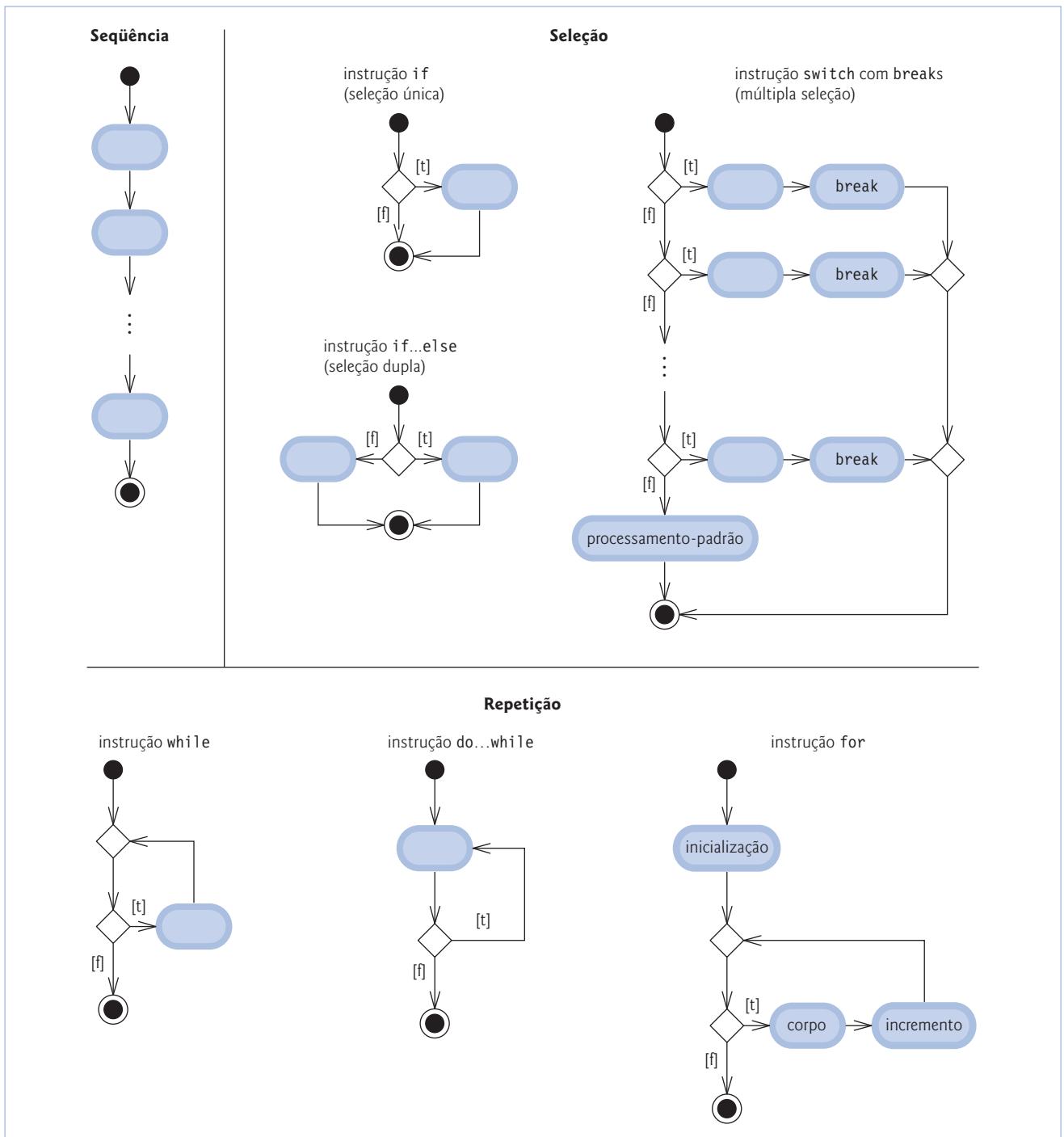
Por simplicidade, são utilizadas apenas instruções de controle de entrada única/saída única — há somente uma maneira de entrar e somente uma maneira de sair de cada instrução de controle. É simples conectar instruções de controle sequencialmente para formar programas estruturados — o estado final de uma instrução de controle é conectado ao estado inicial da próxima instrução de controle — isto é, as instruções de controle são colocadas uma depois da outra em um programa. Chamamos isso de ‘empilhamento de instrução de controle’. As regras para formar programas estruturados também permitem que instruções de controle sejam aninhadas.

A Figura 5.21 mostra as regras para formar programas estruturados. As regras assumem que os estados de ação podem ser utilizados para indicar qualquer ação. As regras também assumem que iniciamos com o chamado diagrama de atividades mais simples (Figura 5.22) consistindo em somente um estado inicial, um estado de ação, um estado final e setas de transição.

Aplicar as regras da Figura 5.21 resulta sempre em um diagrama de atividades com uma aparência organizada dos blocos de construção. Por exemplo, aplicar a Regra 2 repetidamente ao diagrama de atividades mais simples resulta em um diagrama de atividades que contém muitos estados de ação em sequência (Figura 5.23). A Regra 2 gera uma pilha de instruções de controle, então vamos chamá-la de **regra de empilhamento**. [Nota: As linhas tracejadas verticais na Figura 5.23 não fazem parte da UML. Utilizamos essas linhas para separar os quatro diagramas de atividades que demonstram a Regra 2 da Figura 5.21 sendo aplicada.]

A Regra 3 é chamada de **regra de aninhamento**. Aplicar a Regra 3 repetidamente ao diagrama de atividades mais simples resulta em um diagrama de atividades com instruções de controle organizadamente aninhadas. Por exemplo, na Figura 5.24, o estado de ação no diagrama de atividades mais simples é substituído por uma instrução de seleção dupla (`if...else`). Então a Regra 3 é aplicada novamente aos estados de ação na instrução de seleção dupla, substituindo cada um desses estados de ação por uma instrução de seleção dupla. Os símbolos de estado de ação tracejados em torno de cada uma das instruções de seleção dupla representam um estado de ação que foi substituído no diagrama de atividades precedente. [Nota: As setas tracejadas e os símbolos de estado de ação tracejados mostrados na Figura 5.24 não fazem parte da UML. São utilizados aqui como recursos didáticos para ilustrar que qualquer estado de ação pode ser substituído por uma instrução de controle.]

A Regra 4 gera estruturas maiores, mais complexas e mais profundamente aninhadas. Os diagramas que emergem da aplicação das regras na Figura 5.21 constituem o conjunto de todos os possíveis diagramas de atividades e, portanto, o conjunto de todos os programas



**Figura 5.20** As instruções de sequência de entrada única/saída única, seleção e repetição do C++.

### Regras para formar programas estruturados

- 1) Comece com o “diagrama de atividades mais simples” (Figura 5.22).
- 2) Qualquer estado de ação pode ser substituído por dois estados de ação em sequência.
- 3) Qualquer ação pode ser substituída por qualquer instrução de controle (sequência, if, if...else, switch, while, do...while ou for).
- 4) As Regras 2 e 3 podem ser aplicadas com a frequência que você quiser em qualquer ordem.

**Figura 5.21** As regras para formar programas estruturados.

estruturados possíveis. A beleza da abordagem estruturada é que utilizamos apenas sete instruções simples de entrada única/saída única e os montamos de apenas duas maneiras simples.

Se as regras da Figura 5.21 forem seguidas, um diagrama de atividades com sintaxe ilegal (como aquele da Figura 5.25) não pode ser criado. Se você não tiver certeza se um diagrama particular é válido, aplique as regras da Figura 5.21 na ordem inversa para reduzir o diagrama ao diagrama de atividades mais simples. Se for reduzível ao mais simples diagrama de atividades, o diagrama original é estruturado; caso contrário, não é.

A programação estruturada promove a simplicidade. Böhm e Jacopini nos deram o resultado de que apenas três formas de controle são necessárias:

- Sequência
- Seleção
- Repetição

A estrutura de sequência é trivial. Liste simplesmente as instruções para executar na ordem em que elas devem executar.

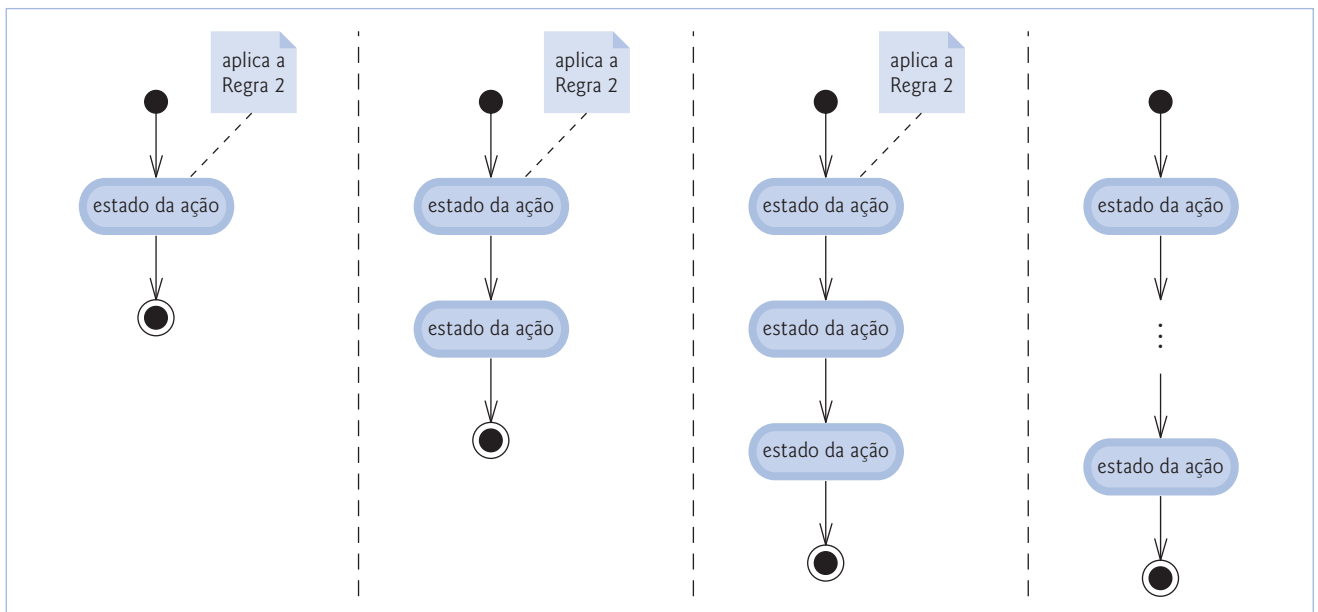
A seleção é implementada de uma destas três maneiras:

- instrução `if` (seleção única)
- instrução `if...else` (seleção dupla)
- instrução `switch` (seleção múltipla)

É simples e direto provar que a instrução simples `if` é suficiente para fornecer qualquer forma de seleção — tudo o que pode ser feito com a instrução `if...else` e a instrução `switch` pode ser implementado (embora talvez não de modo tão claro e eficiente) combinando-se instruções `if`.

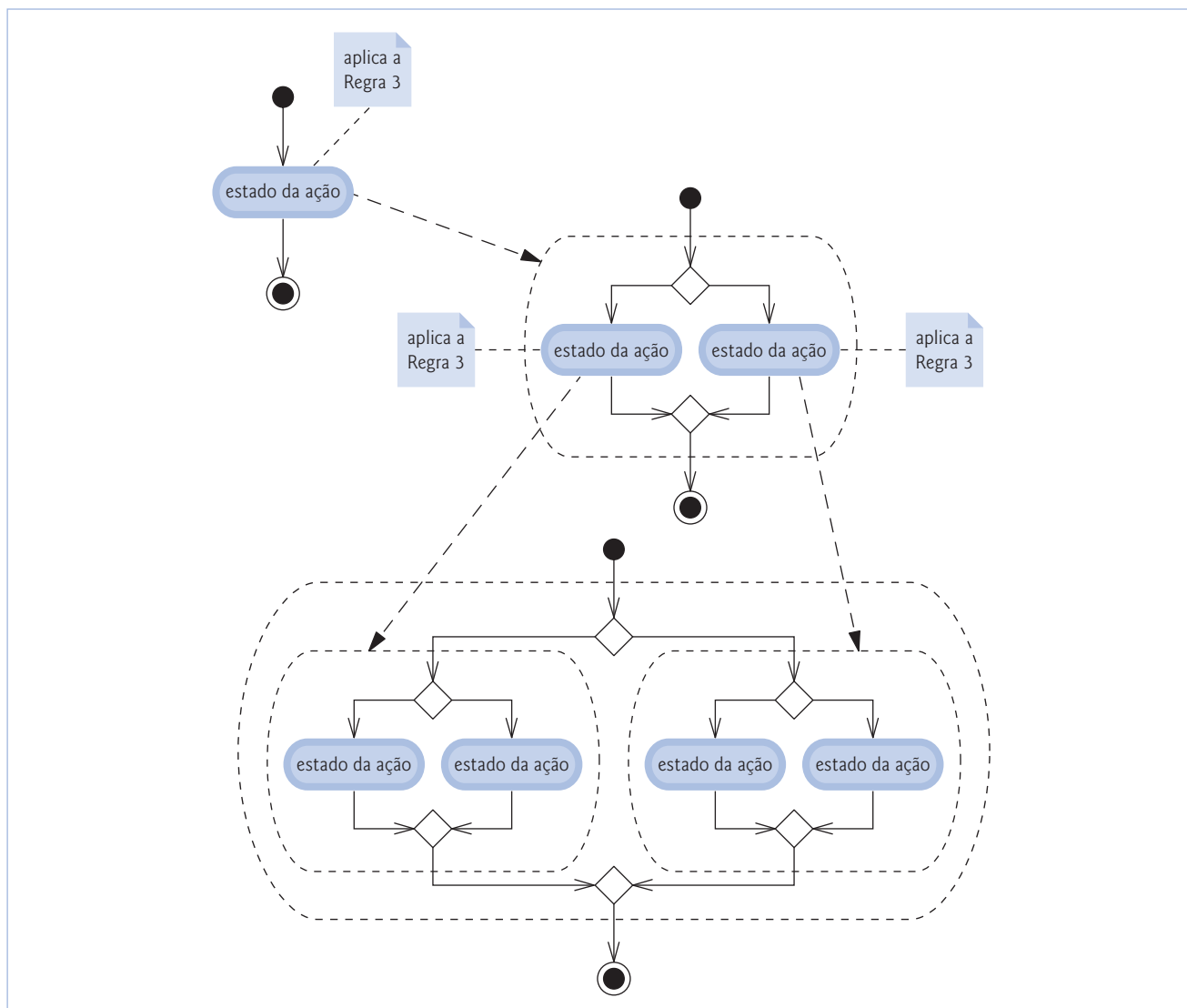


**Figura 5.22** Diagrama de atividades mais simples.

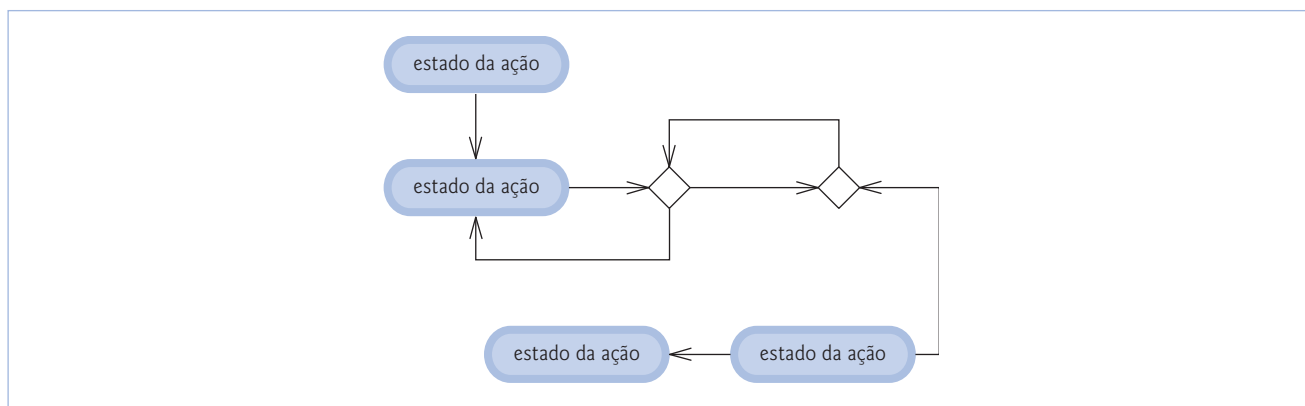


**Figura 5.23** Aplicando repetidamente a Regra 2 da Figura 5.21 ao mais simples diagrama de atividades.





**Figura 5.24** Aplicando várias vezes a Regra 3 da Figura 5.21 ao mais simples diagrama de atividades.



**Figura 5.25** O diagrama de atividades com sintaxe inválida.

A repetição é implementada de uma destas três maneiras:

- Instrução `while`
- Instrução `do...while`
- Instrução `for`

É simples provar que a instrução `while` é suficiente para fornecer qualquer forma de repetição. Tudo o que pode ser feito com a instrução `do...while` e a instrução `for` pode ser feito (embora talvez não tão suavemente) com a instrução `while`.

Combinar esses resultados ilustra que qualquer forma de controle alguma vez necessária em um programa C++ pode ser expressa nos termos a seguir:

- sequência
- instrução `if` (seleção)
- instrução `while` (repetição)

e que essas instruções de controle podem ser combinadas somente de duas maneiras — empilhamento e aninhamento. De fato, a programação estruturada promove simplicidade.

## 5.11 Estudo de caso de engenharia de software: identificando estados e atividades dos objetos no sistema ATM (opcional)

Na Seção 4.13, identificamos muitos dos atributos de classe necessários para implementar o sistema ATM e os adicionamos ao diagrama de classes na Figura 4.24. Nesta seção, mostramos como esses atributos representam o estado de um objeto. Identificamos alguns estados-chave que nossos objetos podem ocupar e discutimos como os objetos mudam de estado em resposta a vários eventos que ocorrem no sistema. Também discutimos o fluxo de trabalho, ou **atividades**, que os objetos realizam no sistema ATM. Nesta seção, apresentamos as atividades dos objetos de transação `BalanceInquiry` e `Withdrawal`, quando eles representam duas das atividades-chave no sistema ATM.

### Diagramas de máquina de estado

Todo objeto em um sistema passa por uma série de estados discretos. O estado atual de um objeto é indicado pelos valores dos atributos do objeto em um dado momento. Os **diagramas de estados de máquina** (comumente chamados de **diagramas de estados**) modelam estados-chave de um objeto e mostram sob que circunstâncias o objeto muda de estado. Ao contrário dos diagramas de classes apresentados nas seções anteriores de estudo de caso, que focalizaram principalmente a estrutura do sistema, os diagramas de estados modelam algum comportamento do sistema.

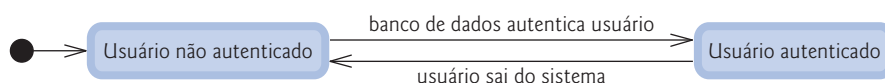
A Figura 5.26 é um diagrama de estados simples que modela alguns estados de um objeto de classe ATM. A UML representa cada estado em um diagrama de estados como um **retângulo arredondado** com o nome do estado posicionado dentro dele. Um **círculo sólido** com uma seta designa o **estado inicial**. Lembre-se de que modelamos essas informações de estado como o atributo `Boolean` `userAuthenticated` no diagrama de classes da Figura 4.24. Esse atributo é inicializado como `false`, ou como o estado ‘Usuário não autenticado’, de acordo com o diagrama de estados.

As setas indicam **transições** entre estados. Um objeto pode transitar de um estado para outro em resposta a vários eventos que ocorrem no sistema. O nome ou descrição do evento que causa uma transição é escrito perto da linha que corresponde à transição. Por exemplo, o objeto ATM muda o estado ‘Usuário não autenticado’ para o estado ‘Usuário autenticado’ depois que o banco de dados autentica o usuário. A partir do documento de requisitos, lembre-se de que o banco de dados autentica um usuário comparando o número de conta e o PIN inseridos pelo usuário com os da conta correspondente no banco de dados. Se o banco de dados indicar que o usuário inseriu um número de conta válido e o PIN correto, o objeto ATM transita para o estado ‘Usuário autenticado’ e muda seu atributo `userAuthenticated` para o valor `true`. Quando o usuário sair do sistema escolhendo a opção ‘saída’ a partir do menu principal, o objeto ATM retorna ao estado ‘Usuário não autenticado’ no estado de preparação para o próximo usuário ATM.



### Observação de engenharia de software 5.3

*Em geral, os engenheiros de software não criam diagramas de estados que mostram cada estado possível e transição de estado para todos os atributos — há simplesmente muitos deles. Os diagramas de estados em geral mostram apenas os estados e transições de estado mais importantes ou complexos.*



**Figura 5.26** Diagrama de estados do objeto ATM.

### Diagramas de atividades

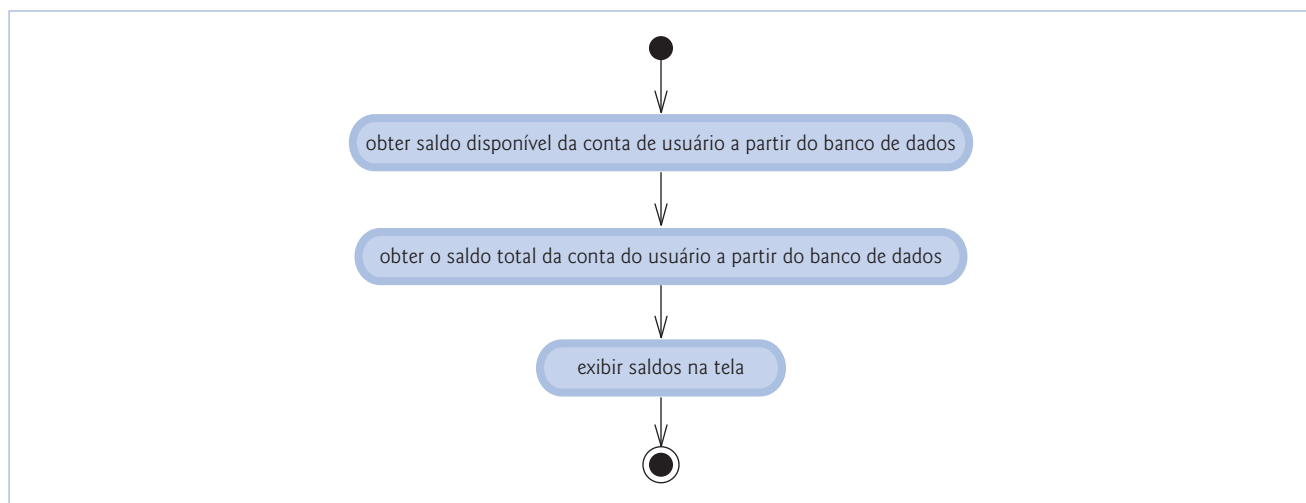
Como um diagrama de estados, um diagrama de atividades modela aspectos do comportamento de sistema. Ao contrário de um diagrama de estados, um diagrama de atividades modela o fluxo de trabalho de um objeto (seqüência de eventos) durante a execução de programa. Um diagrama de atividades modela as ações que o objeto realizará e em que ordem. Lembre-se de que utilizamos os diagramas de atividades UML para ilustrar o fluxo de controle para as instruções de controle apresentadas nos capítulos 4 e 5.

O diagrama de atividades na Figura 5.27 modela as ações envolvidas na execução de uma transação `BalanceInquiry`. Assumimos que um objeto `BalanceInquiry` já foi inicializado e recebeu um número de conta válida (aquela do usuário atual), então o objeto sabe que saldo recuperar. O diagrama inclui as ações que ocorrem depois de o usuário selecionar uma consulta de saldo do menu principal e antes de o ATM retornar o usuário para o menu principal — um objeto `BalanceInquiry` não realiza nem inicia essas ações, então não as modelamos aqui. O diagrama inicia com a recuperação do saldo disponível da conta do usuário a partir do banco de dados. Em seguida, o `BalanceInquiry` recupera o saldo total da conta. Por fim, a transação exibe o saldo na tela. Essa ação completa a execução da transação.

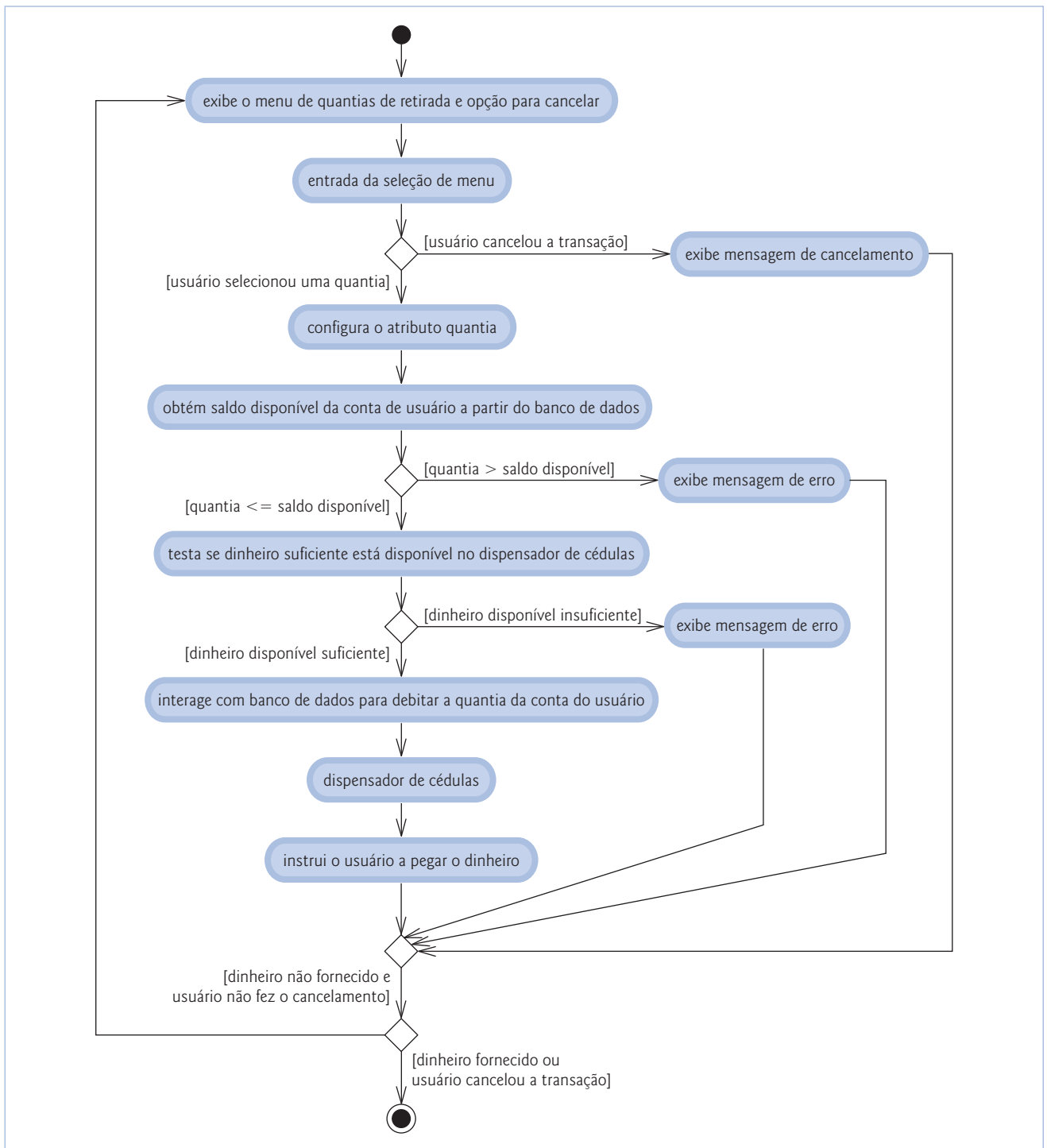
A UML representa uma ação em um diagrama de atividades como um estado de ação modelado por um retângulo com seus lados esquerdo e direito substituídos por arcos que se curvam para fora. Todo estado de ação contém uma expressão de ação — por exemplo, ‘obter saldo disponível da conta do usuário a partir do banco de dados’ — isso especifica uma ação a ser realizada. Uma seta conecta dois estados de ação, indicando a ordem em que ocorrem as ações representadas pelos estados de ação. O círculo sólido (na parte superior da Figura 5.27) representa o estado inicial da atividade — o começo do fluxo de trabalho antes de o objeto realizar as ações modeladas. Nesse caso, a transação primeiro executa a expressão de ação ‘obter saldo disponível da conta do usuário a partir do banco de dados’. E, em seguida, a transação recupera o saldo total. Por fim, a transação exibe ambos os saldos na tela. O círculo sólido dentro de um círculo vazado (na parte inferior da Figura 5.27) representa o estado final — o fim do fluxo de trabalho depois de o objeto realizar as ações modeladas.

A Figura 5.28 mostra um diagrama de atividades de uma transação `Withdrawal`. Assumimos que um número de conta válida foi atribuído ao objeto `Withdrawal`. Não modelamos o usuário que seleciona a opção de saque do menu principal ou a ATM que retorna o usuário para o menu principal porque essas não são ações realizadas por um objeto `Withdrawal`. A transação primeiro exibe um menu de valores-padrão de retirada (Figura 2.17) e uma opção de cancelamento da transação. A transação então realiza a entrada de uma seleção de menu do usuário. O fluxo de atividade agora chega a um símbolo de decisão. Esse ponto determina a próxima ação com base nas condições de guarda associadas. Se o usuário cancelar a transação, o sistema exibe uma mensagem apropriada. Em seguida, o fluxo de cancelamento alcança um símbolo de agregação, em que esse fluxo de atividade se funde com outros possíveis fluxos de atividade da transação (que discutiremos em breve). Observe que uma agregação pode ter qualquer número de setas que entram na transição, mas apenas uma seta que sai da transição. A decisão na parte inferior do diagrama determina se a transação deve repetir desde o início. Quando o usuário tiver cancelado a transação, a condição de guarda ‘dinheiro fornecido ou usuário cancelou a transação’ é verdadeira, então o controle passa para o estado final da atividade.

Se o usuário selecionar uma quantia de saque do menu, a transação configura `amount` (um atributo da classe `Withdrawal` originalmente modelado na Figura 4.24) como o valor escolhido pelo usuário. A próxima transação obtém o saldo disponível da conta do usuário (isto é, o atributo `availableBalance` do objeto `Account` do usuário) a partir do banco de dados. O fluxo de atividade então chega à outra decisão. Se o valor da retirada solicitada exceder o saldo disponível do usuário, o sistema exibe uma mensagem de erro apropriada informando o usuário do problema. O controle então se funde com o outro fluxo de atividades antes de alcançar a decisão na parte inferior do diagrama. A decisão de guarda ‘dinheiro não fornecido e usuário não cancelou a transação’ é verdadeira, então o fluxo da atividades retorna à parte superior do diagrama e a transação solicita ao usuário a entrada de uma nova quantia.



**Figura 5.27** Diagrama de atividades de uma transação `BalanceInquiry`.



**Figura 5.28** Diagrama de atividades de uma transação Withdrawal.

Se a quantia de retirada solicitada for menor que ou igual ao saldo disponível do usuário, a transação testa se o dispensador de dinheiro tem o suficiente para satisfazer a solicitação de retirada. Se não tiver, a transação exibe uma mensagem de erro apropriada e passa pela agregação antes de alcançar a decisão final. O dinheiro não foi fornecido, então o fluxo de atividades retorna ao começo do diagrama de atividades e a transação solicita ao usuário para escolher uma nova quantia. Se dinheiro suficiente estiver disponível, a transação interage com o banco de dados para debitar o valor de retirada da conta de usuário (isto é, subtrair a quantia tanto do atributo `availableBalance` como do `totalBalance` do objeto `Account` do usuário). A transação então libera a quantia de dinheiro desejada e instrui o usuário a pegar o dinheiro que é liberado. O principal fluxo de atividade seguinte funde-se com os dois fluxos de erro e o fluxo de cancelamento. Nesse caso, o dinheiro foi fornecido, portanto o fluxo de atividades alcança o estado final.

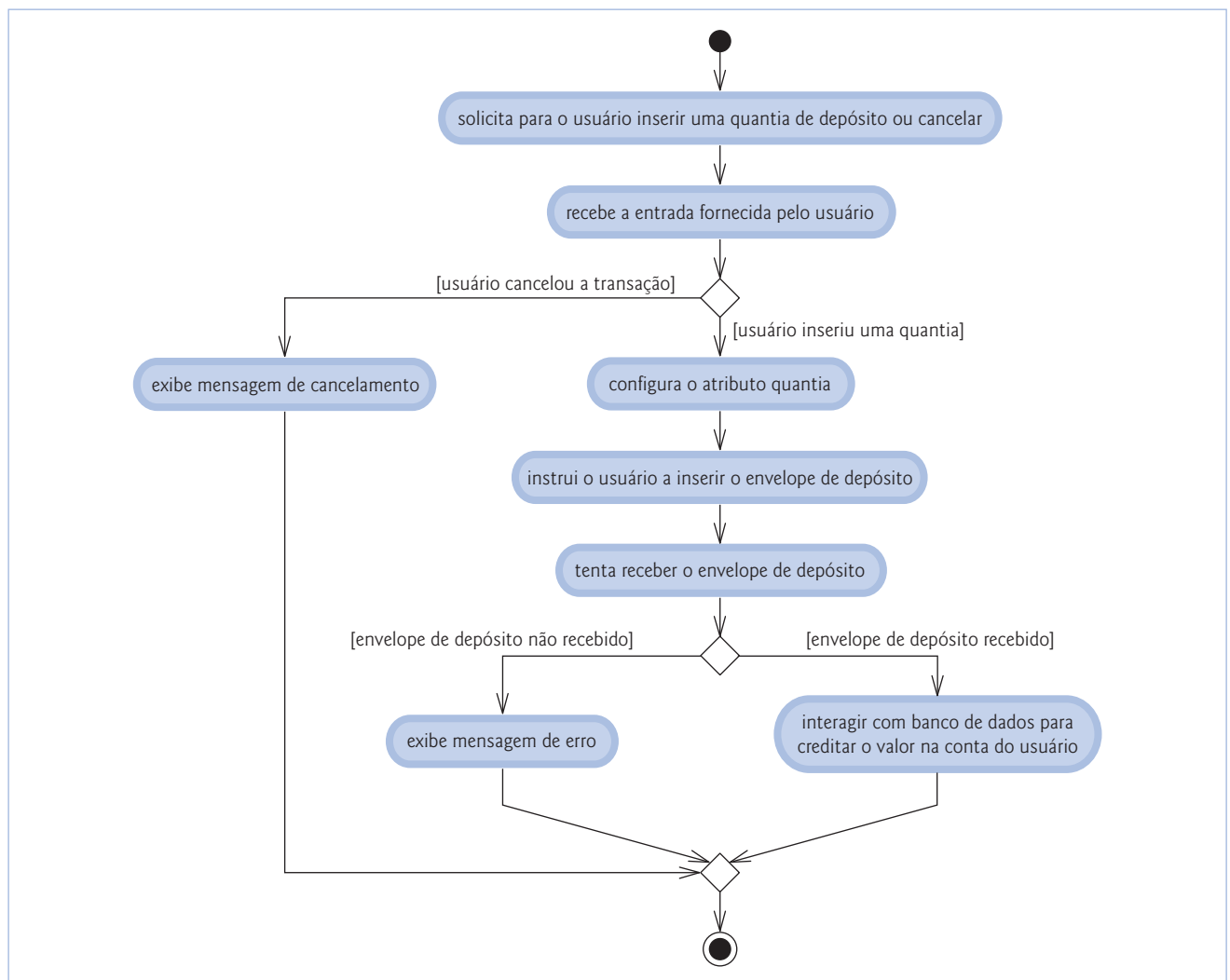
Demos os primeiros passos na modelagem do comportamento do sistema ATM e mostramos como os atributos de um objeto participam da execução das atividades do objeto. Na Seção 6.22, investigamos as operações de nossas classes para criar um modelo mais completo do comportamento do sistema.

### Exercícios de revisão do estudo de caso de engenharia de software

- 5.1** Determine se a seguinte sentença é verdadeira ou falsa e, se falsa, explique por quê: Diagramas de estados modelam aspectos estruturais de um sistema.
- 5.2** Um diagrama de atividades modela os(as) \_\_\_\_\_ que um objeto realiza e a ordem em que ele os(as) realiza.
- ações
  - atributos
  - estados
  - transições de estado
- 5.3** Com base no documento de requisitos, crie um diagrama de atividades para uma transação de depósito.

### Respostas aos exercícios de revisão do estudo de caso de engenharia de software

- 5.1** Falsa. Diagramas de estados modelam algum comportamento de um sistema.
- 5.2** a.
- 5.3** A Figura 5.29 apresenta um diagrama de atividades para uma transação de depósito. O diagrama modela as ações que ocorrem depois de o usuário escolher a opção de depósito no menu principal e antes de o ATM retornar o usuário para o menu principal. Lembre-se de que parte



**Figura 5.29** Diagrama de atividades de uma transação Depósito.

do recebimento de um depósito do usuário envolve converter um número inteiro de centavos em uma quantia em dólar. Lembre-se também de que fazer um depósito em uma conta envolve aumentar apenas o atributo `totalBalance` do objeto `Account` do usuário. O banco só atualiza o atributo `availableBalance` do objeto `Account` do usuário depois de confirmar a quantia em dinheiro no envelope de depósito e, no caso dos cheques, depois da compensação — isso ocorre independentemente do sistema ATM.

## 5.12 Síntese

Neste capítulo, completamos nossa introdução às instruções de controle do C++, que permitem aos programadores controlar o fluxo de execução em funções. O Capítulo 4 discutiu as instruções `if`, `if...else` e `while`. Este capítulo demonstrou as demais instruções de controle do C++ — `for`, `do...while` e `switch`. Mostramos que qualquer algoritmo pode ser desenvolvido utilizando combinações da estrutura de seqüência (isto é, instruções listadas na ordem em que devem executar), os três tipos de instruções de seleção — `if`, `if...else` e `switch` — e os três tipos de instruções de repetição — `while`, `do...while` e `for`. Neste capítulo e no Capítulo 4, discutimos como os programadores podem combinar esses blocos de construção para utilizar as comprovadas técnicas de construção de programa e solução de problemas. Este capítulo também introduziu operadores lógicos do C++, que permitem aos programadores utilizar expressões condicionais mais complexas em instruções de controle. Por fim, examinamos os erros comuns de confundir o operador de igualdade com o de atribuição e fornecemos sugestões para evitar esses erros.

No Capítulo 3, introduzimos a programação C++ com os conceitos básicos de objetos, classes e funções-membro. O Capítulo 4 e este capítulo forneceram uma introdução completa aos tipos de instruções de controle que os programadores geralmente utilizam para especificar lógica do programa em funções. No Capítulo 6, examinamos as funções em maior profundidade.

### Resumo

- Em C++, é preciso chamar uma declaração que também reserva memória de uma definição.
- A instrução de repetição `for` trata todos os detalhes da repetição controlada por contador. O formato geral da instrução `for` é

```
for (inicialização; condiçãoDeContinuaçãoDoLoop; incremento)
 instrução
```

onde *inicialização* inicializa a variável de controle do loop, *condiçãoDeContinuaçãoDoLoop* é a condição que determina se o loop deve continuar executando e *incremento* incrementa a variável de controle.

- Em geral, as instruções `for` são utilizadas para repetição controlada por contador e as instruções `while` são utilizadas para repetição controlada por sentinela.
- O escopo de uma variável define onde ela pode ser utilizada em um programa. Por exemplo, uma variável de controle declarada no cabeçalho de uma instrução `for` só pode ser utilizada no corpo da instrução `for` — a variável de controle não será conhecida fora da instrução `for`.
- As expressões de inicialização e incremento no cabeçalho de uma instrução `for` podem ser listas de expressões separadas por vírgulas. As vírgulas, como utilizadas nessas expressões, são operadores vírgula, que garantem que as listas de expressões sejam avaliadas da esquerda para a direita. O operador vírgula tem a precedência mais baixa de todos os operadores C++. O valor e o tipo de uma lista de expressões separada por vírgulas são o valor e o tipo da expressão mais à direita na lista.
- As expressões de inicialização, condição de continuação do loop e incremento de uma instrução `for` podem conter expressões aritméticas. Além disso, o incremento de uma instrução `for` pode ser negativo, caso em que é realmente um decremento e o loop conta para baixo.
- Se a condição de continuação do loop em um cabeçalho `for` é inicialmente `false`, o corpo da instrução `for` não é executado. Em vez disso, a execução prossegue com a instrução seguinte ao `for`.
- A função de biblioteca-padrão `pow( x, y )` calcula o valor de  $x$  elevado à  $y^{\text{ésima}}$  potência. A função `pow` recebe dois argumentos do tipo `double` e retorna um valor `double`.
- Manipulador de fluxo parametrizado `setw` especifica a largura de campo em que a próxima saída de valor deve aparecer. Por padrão, o valor é alinhado à direita no campo. Se o valor a ser enviado para a saída for maior que a largura de campo, a largura de campo é estendida para acomodar o valor inteiro. O manipulador de fluxo não parametrizado `left` (encontrado no cabeçalho `<iostream>`) pode ser utilizado para fazer com que um valor seja alinhado à esquerda em um campo; e o `right` pode ser utilizado para restaurar o alinhamento à direita.
- Configurações aderentes são aquelas configurações de formatação de saída que permanecem em vigor até serem alteradas.
- A instrução de repetição `do...while` testa a condição de continuação do loop no final do loop, então o corpo do loop será executado pelo menos uma vez. O formato da instrução `do...while` é:

```
do
{
 instrução
} while (condição);
```



- A instrução de seleção múltipla `switch` executa diferentes ações com base nos possíveis valores de uma variável ou expressão. Cada ação é associada com o valor de uma expressão inteira constante (isto é, qualquer combinação de constantes de caractere e constantes inteiras que são avaliadas como um valor constante inteiro) que a variável ou expressão na qual o `switch` é baseado pode assumir.
- A instrução `switch` consiste em uma série de rótulos `case` e um caso `default` opcional.
- A função `cin.get()` lê um caractere do teclado. Os caracteres são normalmente armazenados em variáveis do tipo `char`; entretanto, os caracteres podem ser armazenados em qualquer tipo de dados inteiro, porque são representados como inteiros de 1 byte no computador. Portanto, um caractere pode ser tratado como um inteiro ou como um caractere, dependendo de seu uso.
- O indicador de fim do arquivo é uma combinação de pressionamento de teclas dependente de sistema que especifica que não há mais dados a inserir. EOF (*end-of-file*) é uma constante simbólica inteira definida no arquivo de cabeçalho `<iostream>` que indica ‘fim de arquivo’.
- A expressão entre os parênteses que se seguem à palavra-chave `switch` é chamada de expressão controladora do `switch`. A instrução `switch` compara o valor da expressão de controle a cada rótulo `case`.
- Listar `cases` consecutivamente sem instruções entre eles permite que os `cases` realizem o mesmo conjunto de instruções.
- Cada `case` pode ter múltiplas instruções. A instrução de seleção `switch` difere de outras instruções de controle porque não exige que as múltiplas instruções em cada `case` estejam entre chaves.
- A instrução `switch` pode ser utilizada somente para testar uma expressão inteira constante. Uma constante de caractere é representada como o caractere específico entre aspas simples, como `'A'`. Uma constante de inteiro é simplesmente um valor de inteiro. Além disso, cada rótulo `case` pode especificar apenas uma expressão inteira constante.
- O C++ fornece vários tipos de dados para representar inteiros — `int`, `char`, `short` e `long`. O intervalo de valores inteiros para cada tipo depende do hardware do computador particular.
- A instrução `break`, quando executada em uma das instruções de repetição (`for`, `while` e `do...while`), ocasiona saída imediata da instrução.
- A instrução `continue`, quando executada em uma das instruções de repetição (`for`, `while` e `do...while`), ignora todas as instruções restantes no corpo da instrução de repetição e prossegue com a próxima iteração do loop. Em uma instrução `while` ou `do...while`, a execução continua com a próxima avaliação da condição. Em uma instrução `for`, a execução continua com a expressão de incremento no cabeçalho de instrução `for`.
- Os operadores lógicos permitem aos programadores formar condições complexas combinando condições simples. Os operadores lógicos são `&&` (E lógico), `||` (OU lógico) e `!` (NÃO lógico, também chamado de negação lógica).
- O operador `&&` (E lógico) certifica-se de que duas condições são *ambas* `true` antes de escolher certo caminho de execução.
- O operador `||` (OU lógico) certifica-se de que qualquer uma *ou* ambas as duas condições são `true` antes de escolher certo caminho de execução.
- Uma expressão contendo os operadores `&&` ou `||` é avaliada somente até que a condição de verdade ou falsidade da expressão seja conhecida. Esse recurso de desempenho para avaliação de expressões com E lógico e OU lógico é chamado avaliação de curto-circuito.
- O operador `!` (NÃO lógico, também chamado de negação lógica) permite ao programador ‘inverter’ o significado de uma condição. O operador unário de negação lógica é colocado antes de uma condição escolher um caminho de execução se a condição original (sem o operador de negação lógica) for `false`. Na maioria dos casos, o programador pode evitar a utilização da negação lógica expressando a condição com um operador relacional ou de igualdade apropriado.
- Quando utilizado como uma condição, qualquer valor não-zero é convertido implicitamente em `true`; 0 (zero) é implicitamente convertido em `false`.
- Por padrão, os valores `bool true` e `false` são exibidos por `cout` como 1 e 0, respectivamente. O manipulador de fluxo `boolalpha` especifica que o valor de cada expressão `bool` deve ser exibido como a palavra ‘true’ ou ‘false’.
- Qualquer forma de controle que possa ser necessária um dia em um programa C++ pode ser expressa em termos de instruções de seqüência, seleção e repetição e essas podem ser combinadas somente de duas maneiras — empilhamento e aninhamento.

## Terminologia

!, operador NÃO lógico

&&, operador lógico

||, operador lógico

alinhamento à direita

alinhamento à esquerda

`boolalpha`, manipulador de fluxo

`break`, instrução

`case`, rótulo

`char`, tipo fundamental

condição de continuação do loop

condição simples

configuração aderente

conjunto de caracteres ASCII

`continue`, instrução

curto-circuito, avaliação

decrementar uma variável de controle

`default`, caso, em um `switch`

definição

E lógico (&&)

empilhamento, regra

erro, *off-by-one*

escopo de uma variável

expressão controladora de um `switch`

expressão inteira constante

`for`, cabeçalho

`for`, instrução de repetição

incrementar uma variável de controle

largura de campo

`left`, manipulador de fluxo

loop de retardo

|                                         |                                                     |                                           |
|-----------------------------------------|-----------------------------------------------------|-------------------------------------------|
| <i>lvalue</i> ('left value')            | OU lógico (   )                                     | tabela-verdade                            |
| manipulador de fluxo <code>right</code> | <code>pow</code> , função da biblioteca-padrão      | valor final de uma variável de controle   |
| NÃO lógico (!)                          | regra de aninhamento                                | valor inicial de uma variável de controle |
| negação lógica (!)                      | <i>rvalue</i> ('right value')                       | vírgula, operador                         |
| nome de uma variável de controle        | <code>setw</code> , manipulador de fluxo            | zero, contagem baseada em                 |
| operador lógico                         | <code>switch</code> , instrução de múltipla seleção |                                           |

## Exercícios de revisão

- 5.1** Determine se as seguintes afirmações são *verdadeiras* ou *falsas*. Se a resposta for *falsa*, explicar por quê.
- O caso `default` é requerido na instrução de seleção `switch`.
  - A instrução `break` é requerida no caso-padrão de uma instrução de seleção `switch` para sair do `switch` adequadamente.
  - A expressão `( x > y && a < b )` é `true` se a expressão `x > y` for `true` ou a expressão `a < b` for `true`.
  - Uma expressão contendo o operador `| |` é `true` se qualquer um ou ambos de seus operandos forem `true`.
- 5.2** Escreva uma instrução C++ ou um conjunto de instruções C++ para realizar cada uma das seguintes tarefas:
- Somar os inteiros ímpares entre 1 e 99 utilizando uma instrução `for`. Assumir que as variáveis de inteiro `sum` e `count` foram declaradas.
  - Imprimir o valor 333.546372 em uma largura de campo de 15 caracteres com precisões de 1, 2 e 3. Imprimir cada número na mesma linha. Alinhar à esquerda cada número em seu campo. O que os três valores imprimem?
  - Calcular o valor de 2,5 elevado à potência de 3 utilizando a função `pow`. Imprimir o resultado com uma precisão de 2 em uma largura de campo de 10 posições. O que é impresso?
  - Imprimir os inteiros de 1 a 20 utilizando um loop `while` e a variável de contador `x`. Assumir que a variável `x` foi declarada mas não foi inicializada. Imprimir somente 5 inteiros por linha. [Dica: Utilizar o cálculo `x % 5`. Quando o valor disso for 0, imprimir um caractere de nova linha; caso contrário, imprimir um caractere de tabulação.]
  - Repetir o Exercício 5.2 (d) utilizando uma instrução `for`.
- 5.3** Localize o(s) erro(s) em cada um dos seguintes segmentos de código e explique como corrigi-lo(s).
- ```
x = 1;
while ( x <= 10 );
    x++;
}
```
 - ```
for (y = .1; y != 1.0; y += .1)
 cout << y << endl;
```
  - ```
switch ( n )
{
    case 1:
        cout << "The number is 1" << endl;
    case 2:
        cout << "The number is 2" << endl;
        break;
    default:
        cout << "The number is not 1 or 2" << endl;
        break;
}
```
 - O seguinte código deve imprimir os valores 1 a 10.

```
n = 1;
while ( n < 10 )
    cout << n++ << endl;
```

Respostas dos exercícios de revisão

- 5.1**
- Falsa. O caso `default` é opcional. Se nenhuma ação-padrão é necessária, então não há necessidade de um caso `default`. Contudo, é considerada boa engenharia de software sempre fornecer um caso `default`.
 - Falsa. A instrução `break` é utilizada para sair da instrução `switch`. A instrução `break` não é requerida quando o caso `default` é o último caso. Nem a instrução `break` será necessária se fizer sentido ter o controle prosseguindo com o próximo caso.
 - Falsa. Ao utilizar o operador `&&`, ambas as expressões relacionais devem ser `true` para que a expressão inteira seja `true`.
 - Verdadeira.

5.2

```
a) sum = 0;
   for ( count = 1; count <= 99; count += 2 )
       sum += count;
b) cout << fixed << left
   << setprecision( 1 ) << setw( 15 ) << 333.546372
   << setprecision( 2 ) << setw( 15 ) << 333.546372
   << setprecision( 3 ) << setw( 15 ) << 333.546372
   << endl;
```

A saída é:

```
333.5          333.55          333.546
```

```
c) cout << fixed << setprecision( 2 )
   << setw( 10 ) << pow( 2.5, 3 )
   << endl;
```

A saída é:

```
15.63
```

```
d) x = 1;
```

```
while ( x <= 20 )
{
    cout << x;

    if ( x % 5 == 0 )
        cout << endl;
    else
        cout << '\t';

    x++;
}
```

```
e) for ( x = 1; x <= 20; x++ )
{
    cout << x;

    if ( x % 5 == 0 )
        cout << endl;
    else
        cout << '\t';
}
```

ou

```
for ( x = 1; x <= 20; x++ )
{
    if ( x % 5 == 0 )
        cout << x << endl;
    else
        cout << x << '\t';
}
```

5.3

a) Erro: O ponto-e-vírgula depois do cabeçalho `while` causa um loop infinito.

Correção: Substitua o ponto-e-vírgula por uma `{` ou remova o `;` e a `}`.

b) Erro: Utilizar um número de ponto flutuante para controlar uma instrução de repetição `for`.

Correção: Utilize um inteiro e realize o cálculo adequado a fim de obter os valores que você deseja.

```
for ( y = 1; y != 10; y++ )
    cout << ( static_cast< double >( y ) / 10 ) << endl;
```

c) Erro: Instrução `break` ausente no primeiro `case`.

Correção: Adicione uma instrução `break` ao fim das instruções para o primeiro `case`. Observe que esse não é um erro se o programador quiser que a instrução de `case 2`: execute toda vez que a instrução `case 1`: executar.

d) Erro: Operador relacional impróprio usado na condição de repetição de continuação do `while`.

Correção: Utilize `<=` em vez de `<` ou altere 10 para 11.

Exercícios

5.4 Localize o(s) erro(s) em cada um dos seguintes:

- a) `For ( x = 100, x >= 1, x++ )`
`cout << x << endl;`
- b) O seguinte código deve imprimir se o inteiro `value` for par ou ímpar:
- ```
switch (value % 2)
{
 case 0:
 cout << "Even integer" << endl;
 case 1:
 cout << "Odd integer" << endl;
}
```
- c) O código a seguir deve dar saída dos inteiros ímpares de 19 a 1:
- ```
for ( x = 19; x >= 1; x += 2 )
    cout << x << endl;
```
- d) O código seguinte deve dar saída dos inteiros pares de 2 a 100:
- ```
counter = 2;
```

```
do
{
 cout << counter << endl;
 counter += 2;
} While (counter < 100);
```

**5.5** Escreva um programa que utiliza uma instrução `for` para somar uma sequência de inteiros. Assuma que o primeiro inteiro lido especifica o número de valores que restam a ser inseridos. Seu programa deve ler somente um valor por instrução de entrada. Uma típica sequência de entrada talvez seja

5 100 200 300 400 500

onde o 5 indica que os valores 5 subsequentes devem ser somados.

**5.6** Escreva um programa que utiliza uma instrução `for` para calcular e imprimir a média de vários inteiros. Assuma que o último valor lido é o sentinela 9999. Uma típica sequência de entrada talvez seja

10 8 11 7 9 9999

que indica que o programa deve calcular a média de todos os valores que precedem 9999.

**5.7** O que o seguinte programa faz?

```
1 // Exercício 5.7: ex05_07.cpp
2 // O que esse programa imprime?
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 int main()
9 {
10 int x; // declara x
11 int y; // declara y
12
13 // solicita a entrada do usuário
14 cout << "Enter two integers in the range 1-20: ";
15 cin >> x >> y; // lê valores para x e y
16
17 for (int i = 1; i <= y; i++) // conta de 1 a y
18 {
19 for (int j = 1; j <= x; j++) // conta de 1 a x
20 cout << '@'; // envia @ para saída
21 }
```

```

22 cout << endl; // inicia nova linha
23 } // fim do for externo
24
25 return 0; // indica terminação bem-sucedida
26 } // fim de main

```

- 5.8** Escreva um programa que utiliza uma instrução `for` para localizar o menor de vários inteiros. Assuma que o primeiro valor lido especifica o número de valores restantes e que o primeiro número não é um dos inteiros a ser comparado.
- 5.9** Escreva um programa que utiliza uma instrução `for` para calcular e imprimir o produto dos inteiros ímpares de 1 a 15.
- 5.10** A função fatorial é frequentemente utilizada em problemas de probabilidade. Utilizando a definição de fatorial no Exercício 4.35, escreva um programa que utiliza uma instrução `for` para avaliar o fatorial dos inteiros de 1 a 5. Imprima os resultados no formato de tabela. Que dificuldade poderia impedir você de calcular o fatorial de 20?
- 5.11** Modifique o programa de juros compostos da Seção 5.4 para repetir seus passos para as taxas de juros de 5%, 6%, 7%, 8%, 9% e 10%. Utilize uma instrução `for` para variar a taxa de juros.
- 5.12** Escreva um programa que utiliza instruções `for` para imprimir os seguintes padrões separadamente, um embaixo do outro. Utilize loops `for` para gerar os padrões. Todos os asteriscos (\*) devem ser impressos por uma única instrução na forma `cout << '*'`; (isso faz com que os asteriscos sejam impressos lado a lado). [Dica: Os últimos dois padrões requerem que cada linha inicie com um número apropriado de espaços em branco. *Crédito extra:* Combine seu código dos quatro problemas separados em um único programa que imprime todos os quatro padrões lado a lado usando inteligentemente os loops `for` aninhados.]
- | (a)   | (b)   | (c)   | (d)   |
|-------|-------|-------|-------|
| *     | ***** | ***** | *     |
| **    | ***** | ***** | **    |
| ***   | ***** | ***** | ***   |
| ****  | ***** | ***** | ****  |
| ***** | ***** | ***** | ***** |
| ***** | ***** | ***** | ***** |
| ***** | ****  | ****  | ***** |
| ***** | ***   | ***   | ***** |
| ***** | **    | **    | ***** |
| ***** | *     | *     | ***** |
- 5.13** Uma aplicação interessante dos computadores são os desenhos de gráficos e gráficos de barras. Escreva um programa que lê cinco números (cada um entre 1 e 30). Suponha que o usuário insira apenas valores válidos. Para cada número lido, seu programa deve imprimir uma linha contendo esse número de asteriscos adjacentes. Por exemplo, se seu programa lê o número 7, ele deve imprimir \*\*\*\*\*.
- 5.14** Uma empresa de pedidos pelo correio vende cinco produtos diferentes cujos preços de varejo são: produto 1 — \$ 2,98, produto 2 — \$ 4,50, produto 3 — \$ 9,98, produto 4 — \$ 4,49 e produto 5 — \$ 6,87. Escreva um programa que lê uma série de pares de números como mostrado a seguir:
- a) número de produto  
b) quantidade vendida
- Seu programa deve utilizar uma instrução `switch` para determinar o preço de varejo de cada produto. Seu programa deve calcular e exibir o valor de varejo total de todos os produtos vendidos. Utilize um loop controlado por sentinela para determinar quando o programa deve parar o loop e exibir os resultados finais.
- 5.15** Modifique o programa GradeBook das Figura 5.9–Figura 5.11 para que ele calcule a média de notas baseada em pontos para o conjunto de notas. Uma nota A vale 4 pontos, B vale 3 pontos etc.
- 5.16** Modifique o programa na Figura 5.6 e, então, utilize somente inteiros para calcular os juros compostos. [Dica: Trate todas as quantias monetárias como números inteiros em centavos. Então ‘divida’ o resultado em suas partes dólar e centavos utilizando as operações de divisão e módulo, respectivamente. Insira um ponto.)
- 5.17** Assuma  $i = 1$ ,  $j = 2$ ,  $k = 3$  e  $m = 2$ . O que cada uma das seguintes instruções imprime? Os parênteses são necessários em cada caso?
- a) `cout << ( i == 1 ) << endl;`  
b) `cout << ( j == 3 ) << endl;`  
c) `cout << ( i >= 1 && j < 4 ) << endl;`  
d) `cout << ( m <= 99 && k < m ) << endl;`  
e) `cout << ( j >= i || k == m ) << endl;`  
f) `cout << ( k + m < j || 3 - j >= k ) << endl;`

```
g) cout << (!m) << endl;
h) cout << (!(j - m)) << endl;
i) cout << (!(k > m)) << endl;
```

**5.18** Escreva um programa que imprime uma tabela dos equivalentes binários, octais e hexadecimais dos números decimais no intervalo 1 a 256. Se não estiver familiarizado com esses sistemas de números, leia primeiro o Apêndice D, “Sistemas de numeração”.

**5.19** Calcule o valor de  $\pi$  das séries infinitas

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \dots$$

Imprima uma tabela que mostra o valor aproximado de  $\pi$  depois de cada um dos primeiros 1.000 termos dessa série.

**5.20** (*Triplos de Pitágoras*) Um triângulo retângulo pode ter lados que são todos inteiros. Um conjunto de três valores inteiros para os lados de um triângulo retângulo é chamado de triplo de Pitágoras. Esses três lados devem satisfazer o relacionamento de que a soma dos quadrados de dois dos lados seja igual ao quadrado da hipotenusa. Localize todos os triplos de Pitágoras para `side1`, `side2` e `hypotenuse`, todos não maiores que 500. Utilize um loop `for` triplamente aninhado que tente todas as possibilidades. Esse é um exemplo de computação baseada na **força bruta**. Você aprenderá em cursos mais avançados de ciência da computação que há muitos problemas interessantes para os quais não há nenhuma abordagem algorítmica conhecida diferente da pura força bruta.

**5.21** Uma empresa paga seus empregados como gerentes (que recebem um salário fixo por semana), horistas (que recebem um salário-hora fixo pelas primeiras 40 horas trabalhadas e mais hora extra com 50% de acréscimo, isto é, 1,5 vez seu salário-hora, para as horas extras trabalhadas), comissionados (que recebem \$ 250 mais 5,7% bruto das vendas semanais) ou trabalhadores por produção (que recebem uma quantia fixa de dinheiro para cada item que produzem — cada trabalhador por produção trabalha apenas em um tipo de item nessa empresa). Escreva um programa para computar o pagamento semanal de cada empregado. Você não sabe antecipadamente o número de empregados. Cada tipo de empregado tem seu próprio código de salário: os gerentes têm código 1, os horistas têm código 2, os comissionados têm código 3 e os trabalhadores por produção têm código 4. Utilize um `switch` para calcular o salário de cada empregado de acordo com o código de pagamento desse empregado. Dentro do `switch`, solicite que o usuário (isto é, o caixa que faz a folha de pagamento) insira os fatos apropriados de que o programa precisa para calcular o salário de cada empregado de acordo com o código de pagamento desse empregado.

**5.22** (*Leis de De Morgan*) Neste capítulo, discutimos os operadores lógicos `&&`, `||` e `!`. As leis de De Morgan às vezes podem tornar mais conveniente para nós expressarmos uma expressão lógica. Essas leis afirmam que a expressão `!( condição1 && condição2 )` é logicamente equivalente à expressão `( !condição1 || !condição2 )`. Além disso, a expressão `!( condição1 || condição2 )` é logicamente equivalente à expressão `( !condição1 && !condição2 )`. Utilize as leis de De Morgan para escrever expressões equivalentes para cada uma das seguintes, e então escreva um programa para mostrar que a expressão original e a nova expressão em cada caso são equivalentes:

- `!( x < 5 ) && !( y >= 7 )`
- `!( a == b ) || !( g != 5 )`
- `!( ( x <= 8 ) && ( y > 4 ) )`
- `!( ( i > 4 ) || ( j <= 6 ) )`

**5.23** Escreva um programa que imprime a seguinte forma de losango. Você pode utilizar instruções de saída que imprimem um único asterisco (\*) ou um único espaço em branco. Maximize sua utilização de repetição (com instruções `for` aninhadas) e minimize o número de instruções de saída.

```

 *

 *
```

**5.24** Modifique o programa que você escreveu no Exercício 5.23 para ler um número ímpar no intervalo de 1 a 19 a fim de especificar o número de linhas no losango e, então, exiba um losango do tamanho apropriado.

**5.25** Uma crítica às instruções `break` e `continue` é que elas não são instruções estruturadas. De fato, elas podem ser sempre substituídas por instruções estruturadas, embora possa ser inconveniente fazer isso. Descreva de maneira geral como você removeria qualquer instrução `break` de um loop em um programa e a substituiria por alguma equivalente estruturada. [Dica: A instrução `break` deixa um loop a partir de dentro do corpo do loop. Outra maneira de deixar é fazendo falhar o teste de continuação do loop. Considere a possibilidade de utilizar



no teste de continuação do loop um segundo teste que indica ‘saída prévia por causa de uma condição ‘break’’.] Utilize a técnica que você desenvolveu aqui para remover a instrução `break` do programa da Figura 5.13.

**5.26** O que o seguinte segmento de programa faz?

```

1 for (int i = 1; i <= 5; i++)
2 {
3 for (int j = 1; j <= 3; j++)
4 {
5 for (int k = 1; k <= 4; k++)
6 cout << '*';
7
8 cout << endl;
9 } // fim do for interno
10
11 cout << endl;
12 } // fim do for externo

```

**5.27** Descreva de maneira geral como você removeria qualquer instrução `continue` de um loop em um programa e a substituiria por alguma equivalente estruturada. Utilize a técnica que você desenvolveu aqui para remover a instrução `continue` do programa da Figura 5.14.

**5.28** (A canção ‘*The Twelve Days of Christmas*’) Escreva um programa que utiliza instruções de repetição `switch` para imprimir a canção *The Twelve Days of Christmas*. Uma instrução `switch` deve ser utilizada para imprimir o dia (isto é, ‘First’, ‘Second’ etc.) Uma instrução `switch` separada deve ser utilizada para imprimir o restante de cada verso. Visite o site Web [www.12days.com/library/carols/12daysofxmas.htm](http://www.12days.com/library/carols/12daysofxmas.htm) para obter a letra completa da canção.

**5.29** (*Problema de Peter Minuit*) Diz a lenda que, em 1626, Peter Minuit comprou a Ilha de Manhattan por \$ 24,00 na base da troca. Será que ele fez um bom investimento? Para responder a essa pergunta, modifique o programa de juros compostos da Figura 5.6 para iniciar com um capital de \$ 24,00 e calcular o valor dos juros em depósito se esse dinheiro continuasse depositado até este ano (por exemplo, 379 anos até 2005). Coloque o loop `for` que realiza o cálculo de juros compostos em um loop `for` externo que varia a taxa de juros de 5% a 10% para observar as maravilhas dos juros compostos.



*A forma nunca segue a função.*  
Louis Henri Sullivan

*E pluribus unum.*  
(Um composto de muitos.)  
Virgílio

*Chama o dia de ontem, faz que o tempo atrás retorne.*  
William Shakespeare

*Chamem-me Ismael.*  
Herman Melville

*Quando você me chamar assim, sorria!*  
Owen Wister

*Responda-me em uma palavra.*  
William Shakespeare

*Há um ponto em que os métodos se autodevoram.*  
Frantz Fanon

*A vida só pode ser compreendida olhando-se para trás; mas só pode ser vivida olhando-se para a frente.*  
Soren Kierkegaard

## Funções e uma introdução à recursão

### OBJETIVOS

Neste capítulo, você aprenderá:

- A construir programas modularmente a partir de funções.
- A utilizar funções de matemática comuns disponíveis na C++ Standard Library.
- A criar funções com múltiplos parâmetros.
- Os mecanismos para passar informações entre funções e retornar resultados.
- Como o mecanismo de chamada/retorno de função é suportado pela pilha de chamadas de função e os registros de ativação.
- A utilizar a geração de números aleatórios para implementar aplicativos de jogos.
- Como a visibilidade de identificadores é limitada a regiões específicas de programas.
- A escrever e utilizar funções recursivas, isto é, funções que chamam a si mesmas.

- 6.1 Introdução
- 6.2 Componentes de um programa em C++
- 6.3 Funções da biblioteca de matemática
- 6.4 Definições de funções com múltiplos parâmetros
- 6.5 Protótipos de funções e coerção de argumentos
- 6.6 Arquivos de cabeçalho da biblioteca-padrão C++
- 6.7 Estudo de caso: geração de números aleatórios
- 6.8 Estudo de caso: jogo de azar e introdução a enum
- 6.9 Classes de armazenamento
- 6.10 Regras de escopo
- 6.11 Pilha de chamadas de função e registros de ativação
- 6.12 Funções com listas de parâmetro vazias
- 6.13 Funções inline
- 6.14 Referências e parâmetros de referência
- 6.15 Argumentos-padrão
- 6.16 Operador de solução de escopo unário
- 6.17 Sobrecarga de funções
- 6.18 Templates de funções
- 6.19 Recursão
- 6.20 Exemplo que utiliza recursão: série de Fibonacci
- 6.21 Recursão *versus* iteração
- 6.22 Estudo de caso de engenharia de software: identificando operações de classe no sistema ATM (opcional)
- 6.23 Síntese

Resumo | Terminologia | Exercícios de revisão | Respostas dos exercícios de revisão | Exercícios

## 6.1 Introdução

A maioria dos programas de computador que resolvem problemas do mundo real é bem maior que os programas apresentados nos capítulos iniciais deste livro. A experiência tem mostrado que a melhor maneira de desenvolver e manter um programa grande é construí-lo a partir de pequenas e simples partes, ou componentes. Essa técnica se chama **dividir para conquistar**. Introduzimos funções (como partes de programa) no Capítulo 3. Neste capítulo, estudamos funções em mais profundidade. Enfatizamos como declarar e utilizar funções para facilitar o projeto, a implementação, a operação e a manutenção de grandes programas.

Apresentaremos a visão geral de uma parte das funções de matemática da C++ Standard Library, mostrando as diversas funções que requerem mais de um parâmetro. Em seguida, você aprenderá a declarar uma função com mais de um parâmetro. Também apresentaremos informações adicionais sobre protótipos de função e como o compilador os utiliza para converter o tipo de um argumento em uma chamada de função para o tipo especificado em uma lista de parâmetros da função, se necessário.

Em seguida, faremos uma breve digressão sobre as técnicas de simulação com a geração de números aleatórios (ou randômicos) e desenvolveremos uma versão do jogo de dados de cassino chamado *craps* que utiliza a maioria das técnicas de programação aprendidas até agora no livro.

A seguir, apresentaremos as classes de armazenamento e as regras de escopo do C++. Elas determinam o período em que um objeto existe na memória e onde seu identificador pode ser referenciado em um programa. Você também aprenderá como o C++ é capaz de monitorar as funções que estão atualmente em execução, como os parâmetros e outras variáveis de funções locais são mantidos na memória, e como uma função sabe onde retornar depois de completar a execução. Discutimos dois tópicos que ajudam a melhorar o desempenho do programa — funções inline que podem eliminar o overhead de uma chamada de função e parâmetros de referência que podem ser utilizados para passar grandes itens de dados para funções eficientemente.

Muitos dos aplicativos que você desenvolve terão mais de uma função com o mesmo nome. Essa técnica, chamada sobrecarga de função, é utilizada pelos programadores para implementar as funções que realizam tarefas semelhantes para argumentos de tipos diferentes ou possivelmente para diferentes números de argumentos. Consideramos templates de função — um mecanismo para definir uma família de funções sobrecarregadas. O capítulo encerra com uma discussão de funções que chamam a si próprias, direta ou indiretamente (por outra função) — um tópico chamado recursão que é discutido extensamente em cursos de nível superior de ciência da computação.

## 6.2 Componentes de um programa em C++

Em geral, os programas C++ são escritos combinando novas funções e classes que o programador escreve com funções ‘pré-empacotadas’ e classes disponíveis na C++ Standard Library. Neste capítulo, vamos nos concentrar nas funções.

A C++ Standard Library fornece uma rica coleção de funções para realizar cálculos matemáticos comuns, manipulações de string, manipulações de caractere, entrada/saída, verificação de erros e muitas outras operações úteis. Isso torna o trabalho do programador mais fácil porque essas funções fornecem muitas das capacidades de que o programador precisa. As funções da C++ Standard Library são fornecidas como parte do ambiente de programação C++.



### Observação de engenharia de software 6.1

*Leia a documentação do seu compilador para se familiarizar com as funções e classes da C++ Standard Library.*

As funções (chamadas **métodos** ou **procedimentos** [ou ainda **procedures**] em outras linguagens de programação) permitem que o programador modularize um programa separando suas tarefas em unidades autocontidas. Você utilizou funções em todos os programas que escreveu. Essas funções são às vezes referidas como **funções definidas pelo usuário** ou **funções definidas pelo programador**. As instruções no corpo das funções são escritas apenas uma vez, talvez reutilizadas a partir de diversas localizações em um programa e ocultadas de outras funções.

Há várias motivações para modularizar um programa com funções. Uma delas é a abordagem de dividir e conquistar, que torna o desenvolvimento de programas mais gerenciável, possibilitando que eles sejam construídos a partir de fragmentos simples. Outra é a reusabilidade de software — utilizar funções existentes como blocos de construção para criar novos programas. Por exemplo, nos primeiros programas, não tínhamos de definir como ler uma linha de texto a partir do teclado — o C++ fornece essa capacidade por meio da função `getline` do arquivo de cabeçalho `<string>`. Um terceiro motivo é evitar a repetição de código. Além disso, dividir um programa em funções significativas torna o programa mais fácil de depurar e manter.



### Observação de engenharia de software 6.2

*Para promover a capacidade de reutilização de software, todas as funções devem ser limitadas à realização de uma única tarefa bem definida e o nome da função deve expressar essa tarefa efetivamente. Essas funções tornam os programas mais fáceis de escrever, testar, depurar e manter.*



### Dica de prevenção de erro 6.1

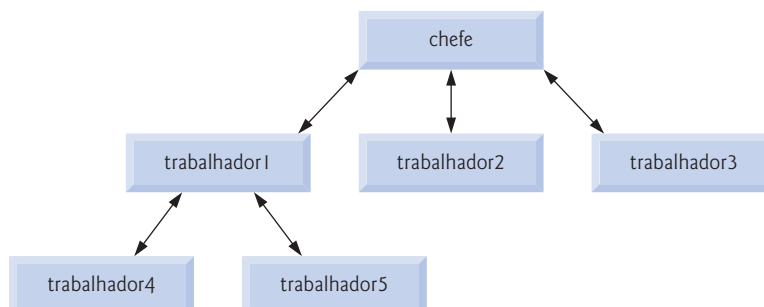
*Uma pequena função que realiza uma tarefa é mais fácil de testar e depurar do que uma função maior que realiza muitas tarefas.*



### Observação de engenharia de software 6.3

*Se você não puder escolher um nome conciso que expresse a tarefa de uma função, sua função pode estar tentando realizar um número excessivo de tarefas. Em geral, é melhor dividir essa função em várias funções menores.*

Como você sabe, uma função é invocada por uma chamada de função e, quando a função chamada completa sua tarefa, ela retorna um resultado ou simplesmente retorna o controle para o chamador. Uma analogia a essa estrutura de programa é a forma hierárquica de gerenciamento (Figura 6.1). Um chefe (semelhante à função chamadora) solicita que um trabalhador (semelhante à função chamada) realize uma tarefa e informe (isto é, retorne) os resultados depois de completar a tarefa. A função-chefe não sabe como a função-trabalhador realiza as tarefas designadas. O trabalhador também pode chamar outras funções-trabalhador, sem que o chefe saiba. Esse ocultamento dos detalhes da implementação promove a boa engenharia de software. A Figura 6.1 mostra a função chefe comunicando-se com várias



**Figura 6.1** Relacionamento hierárquico entre a função-chefe e as funções-trabalhador.

funções-trabalhador de maneira hierárquica. A função-chefe divide as responsabilidades entre as várias funções-trabalhador. Observe que `trabalhador1` atua como uma ‘função-chefe’ para `trabalhador4` e `trabalhador5`.

### 6.3 Funções da biblioteca de matemática

Como você sabe, uma classe pode fornecer funções-membro que realizam os serviços da classe. Por exemplo, nos capítulos 3–5, você chamou as funções-membro de várias versões de um objeto `GradeBook` para exibir a mensagem de boas-vindas do `GradeBook`, configurar seu nome do curso, obter um conjunto de notas e calcular a média dessas notas.

Às vezes as funções não são membros de uma classe. Essas funções são chamadas de **funções globais**. Semelhantemente às funções-membro de uma classe, os protótipos de função para funções globais são colocados em arquivos de cabeçalho, para que as funções globais possam ser reutilizadas em qualquer programa que inclua o arquivo de cabeçalho e possa ser linkado ao código-objeto da função. Por exemplo, lembre-se de que utilizamos a função `pow` do arquivo de cabeçalho `<cmath>` para elevar um valor a uma potência na Figura 5.6. Introduzimos aqui várias funções a partir do arquivo de cabeçalho `<cmath>` para apresentar o conceito de funções globais que não pertencem a uma classe particular. Neste e nos capítulos subseqüentes, utilizamos uma combinação de funções globais (como `main`) e classes com funções-membro para implementar nossos programas de exemplo.

O arquivo de cabeçalho `<cmath>` fornece uma coleção de funções que permite realizar cálculos matemáticos comuns. Por exemplo, você pode calcular a raiz quadrada de 900.0 com a chamada de função

```
sqrt(900.0)
```

A expressão anterior é avaliada como 30.0. A função `sqrt` aceita um argumento de tipo `double` e retorna um resultado `double`. Observe que não há necessidade de criar qualquer objeto antes de chamar a função `sqrt`. Observe também que *todas* as funções no arquivo de cabeçalho `<cmath>` são funções globais — assim, cada uma é chamada simplesmente especificando o nome da função seguido por parênteses contendo os argumentos da função.

Os argumentos de função podem ser constantes, variáveis ou expressões mais complexas. Se `c = 13.0`, `d = 3.0` e `f = 4.0`, então a instrução

```
cout << sqrt(c + d * f) << endl;
```

calcula e imprime a raiz quadrada de  $13.0 + 3.0 * 4.0 = 25.0$  — a saber, 5.0. Algumas funções de biblioteca de matemática são resumidas na Figura 6.2. Na figura, as variáveis `x` e `y` são do tipo `double`.

| Função                    | Descrição                                               | Exemplo                                                                                               |
|---------------------------|---------------------------------------------------------|-------------------------------------------------------------------------------------------------------|
| <code>ceil( x )</code>    | arredonda $x$ para o menor inteiro não menor que $x$    | <code>ceil( 9.2 )</code> é 10.0<br><code>ceil( -9.8 )</code> é -9.0                                   |
| <code>cos( x )</code>     | co-seno trigonométrico de $x$ ( $x$ em radianos)        | <code>cos( 0.0 )</code> é 1.0                                                                         |
| <code>exp( x )</code>     | função exponencial $e^x$                                | <code>exp( 1.0 )</code> é 2.71828<br><code>exp( 2.0 )</code> é 7.38906                                |
| <code>fabs( x )</code>    | valor absoluto de $x$                                   | <code>fabs( 5.1 )</code> é 5.1<br><code>fabs( 0.0 )</code> é 0.0<br><code>fabs( -8.76 )</code> é 8.76 |
| <code>floor( x )</code>   | arredonda $x$ para o maior inteiro não maior que $x$    | <code>floor( 9.2 )</code> é 9.0<br><code>floor( -9.8 )</code> é -10.0                                 |
| <code>fmod( x, y )</code> | resto de $x/y$ como um número de ponto flutuante        | <code>fmod( 2.6, 1.2 )</code> é 0.2                                                                   |
| <code>log( x )</code>     | logaritmo natural de $x$ (base $e$ )                    | <code>log( 2.718282 )</code> é 1.0<br><code>log( 7.389056 )</code> é 2.0                              |
| <code>log10( x )</code>   | logaritmo de $x$ (base 10)                              | <code>log10( 10.0 )</code> é 1.0<br><code>log10( 100.0 )</code> é 2.0                                 |
| <code>pow( x, y )</code>  | $x$ elevado à potência $y$ ( $x^y$ )                    | <code>pow( 2, 7 )</code> é 128<br><code>pow( 9, .5 )</code> é 3                                       |
| <code>sin( x )</code>     | seno trigonométrico de $x$ ( $x$ em radianos)           | <code>sin( 0.0 )</code> é 0                                                                           |
| <code>sqrt( x )</code>    | raiz quadrada de $x$ (onde $x$ é um valor não negativo) | <code>sqrt( 9.0 )</code> é 3.0                                                                        |
| <code>tan( x )</code>     | tangente trigonométrica de $x$ ( $x$ em radianos)       | <code>tan( 0.0 )</code> é 0                                                                           |

Figura 6.2 Funções da biblioteca de matemática.

## 6.4 Definições de funções com múltiplos parâmetros

Os capítulos 3 a 5 apresentaram as classes com funções simples que tinham no máximo um parâmetro. As funções costumam exigir mais de uma informação para realizar suas tarefas. Consideramos agora funções com múltiplos parâmetros.

O programa nas figuras 6.3–6.5 modifica nossa classe `GradeBook` incluindo uma chamada de função definida pelo usuário `maximum` que determina e retorna o maior de três valores `int`. Quando o aplicativo inicia a execução, a função `main` (linhas 5–14 da Figura 6.5) cria um objeto da classe `GradeBook` (linha 8) e chama a função-membro do objeto `inputGrades` (linha 11) para ler três notas do tipo inteiro fornecidas pelo usuário. No arquivo de implementação da classe `GradeBook` (Figura 6.4), as linhas 54–55 da função-membro `inputGrades` solicitam para o usuário inserir três valores do tipo inteiro e os lê a partir do usuário. A linha 58 chama a função-membro `maximum` (definida nas linhas 62–75). A função `maximum` determina o maior valor e, em seguida, a instrução `return` (linha 74) retorna esse valor para o ponto em que a função `inputGrades` invocou `maximum` (linha 58). A função-membro `inputGrades` então armazena valor de retorno de `maximum` no membro de dados `maximumGrade`. Esse valor é então enviado para a saída chamando a função `displayGradeReport` (linha 12 da Figura 6.5). [Nota: Chamamos essa função de `displayGradeReport` porque as versões subsequentes da classe `GradeBook` utilizarão essa função para exibir um relatório de notas completo, incluindo notas máximas e mínimas.] No Capítulo 7, “Arrays e vetores”, aprimoraremos o `GradeBook` para processar um número arbitrário de notas.

```

1 // Figura 6.3: GradeBook.h
2 // Definição de classe GradeBook que localiza a máxima de três notas.
3 // As funções-membro são definidas no GradeBook.cpp
4 #include <string> // o programa utiliza classe de string padrão C++
5 using std::string;
6
7 // definição da classe GradeBook
8 class GradeBook
9 {
10 public:
11 GradeBook(string); // o construtor inicializa o nome do curso
12 void setCourseName(string); // função para configurar o nome do curso
13 string getCourseName(); // função para recuperar o nome do curso
14 void displayMessage(); // exibe uma mensagem de boas-vindas
15 void inputGrades(); // insere três notas fornecidas pelo usuário
16 void displayGradeReport(); // exibe um relatório baseado nas notas
17 int maximum(int, int, int); // determina o máximo de 3 valores
18 private:
19 string courseName; // nome do curso para esse GradeBook
20 int studentMaximum; // máxima de três notas
21 }; // fim da classe GradeBook

```

Figura 6.3 Arquivo de cabeçalho `GradeBook`.

```

1 // Figura 6.4: GradeBook.cpp
2 // Definições de função-membro para a classe GradeBook que
3 // determina a máxima de três notas.
4 #include <iostream>
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include "GradeBook.h" // inclui a definição de classe GradeBook
10
11 // construtor inicializa courseName com string fornecida como argumento;
12 // inicializa studentMaximum como 0
13 GradeBook::GradeBook(string name)

```

Figura 6.4 Classe `GradeBook` define a função `maximum`.

(continua)



```

14 {
15 setCourseName(name); // valida e armazena courseName
16 studentMaximum = 0; // esse valor será substituído pela nota máxima
17 } // fim do construtor GradeBook
18
19 // função para configurar o nome do curso; limita o nome a 25 ou menos caracteres
20 void GradeBook::setCourseName(string name)
21 {
22 if (name.length() <= 25) // se o nome tiver 25 ou menos caracteres
23 courseName = name; // armazena o nome do curso no objeto
24 else // se o nome tiver mais que 25 caracteres
25 { // configura courseName como os primeiros 25 caracteres do nome de parâmetro
26 courseName = name.substr(0, 25); // seleciona os primeiros 25 caracteres
27 cout << "Name \"" << name << "\" exceeds maximum length (25).\n"
28 << "Limiting courseName to first 25 characters.\n" << endl;
29 } // fim do if...else
30 } // fim da função setCourseName
31
32 // função para recuperar o nome do curso
33 string GradeBook::getCourseName()
34 {
35 return courseName;
36 } // fim da função getCourseName
37
38 // exibe uma mensagem de boas-vindas para o usuário de GradeBook
39 void GradeBook::displayMessage()
40 {
41 // essa instrução chama getCourseName para obter o
42 // nome do curso que esse GradeBook representa
43 cout << "Welcome to the grade book for\n" << getCourseName() << "!\n"
44 << endl;
45 } // fim da função displayMessage
46
47 // insere três notas a partir do usuário; determina a máxima
48 void GradeBook::inputGrades()
49 {
50 int grade1; // primeira nota inserida pelo usuário
51 int grade2; // segunda nota inserida pelo usuário
52 int grade3; // terceira nota inserida pelo usuário
53
54 cout << "Enter three integer grades: ";
55 cin >> grade1 >> grade2 >> grade3;
56
57 // armazena máxima no membro studentMaximum
58 studentMaximum = maximum(grade1, grade2, grade3);
59 } // fim da função inputGrades
60
61 // retorna o máximo dos seus três parâmetros inteiros
62 int GradeBook::maximum(int x, int y, int z)
63 {
64 int maximumValue = x; // supõe que x é o maior valor inicial
65
66 // determina se y é maior que maximumValue
67 if (y > maximumValue)
68 maximumValue = y; // torna y o novo maximumValue
69
70 // determina se z é maior que maximumValue

```

Figura 6.4 Classe GradeBook define a função maximum.

(continua)

```

71 if (z > maximumValue)
72 maximumValue = z; // torna z o novo maximumValue
73
74 return maximumValue;
75 } // fim da função maximum
76
77 // exibe um relatório baseado nas notas inseridas pelo usuário
78 void GradeBook::displayGradeReport()
79 {
80 // gera a saída da nota máxima entre as notas inseridas
81 cout << "Maximum of grades entered: " << studentMaximum << endl;
82 } // fim da função displayGradeReport

```

**Figura 6.4** Classe GradeBook define a função maximum.

(continuação)

```

1 // Figura 6.5: fig06_05.cpp
2 // Cria o objeto GradeBook, insere notas e exibe relatório de notas.
3 #include "GradeBook.h" // inclui a definição de classe GradeBook
4
5 int main()
6 {
7 // cria objeto GradeBook
8 GradeBook myGradeBook("CS101 C++ Programming");
9
10 myGradeBook.displayMessage(); // exibe a mensagem de boas-vindas
11 myGradeBook.inputGrades(); // lê as notas fornecidas pelo usuário
12 myGradeBook.displayGradeReport(); // exibe relatório baseado em notas
13 return 0; // indica terminação bem-sucedida
14 } // fim de main

```

Welcome to the grade book for  
CS101 C++ Programming!

Enter three integer grades: **86 67 75**  
Maximum of grades entered: 86

Welcome to the grade book for  
CS101 C++ Programming!

Enter three integer grades: **67 86 75**  
Maximum of grades entered: 86

Welcome to the grade book for  
CS101 C++ Programming!

Enter three integer grades: **67 75 86**  
Maximum of grades entered: 86

**Figura 6.5** Demonstrando a função maximum.



## Observação de engenharia de software 6.4

As vírgulas utilizadas na linha 58 da Figura 6.4 para separar os argumentos para a função maximum não são operadores vírgula como discutido na Seção 5.3. O operador vírgula garante que seus operandos sejam avaliados da esquerda para a direita. Entretanto, a ordem de avaliação dos argumentos de uma função não é especificada pelo padrão C++. Portanto, os diferentes compiladores

*podem avaliar argumentos de função em ordens diferentes. O padrão C++ garante que todos os argumentos em uma chamada de função sejam avaliados antes de a função chamada executar.*



### Dica de portabilidade 6.1

*Às vezes, quando os argumentos de uma função são expressões mais complexas, como aquelas com chamadas para outras funções, a ordem em que o compilador avalia os argumentos poderia afetar os valores de um ou mais dos argumentos. Se a ordem de avaliação muda entre compiladores, os valores de argumento passados para a função poderiam variar, causando erros de lógica sutis.*



### Dica de prevenção de erro 6.2

*Se você tiver dúvidas quanto à ordem de avaliação dos argumentos de uma função e se a ordem poderia afetar ou não os valores passados à função, avalie os argumentos em instruções de atribuição separadas antes da chamada de função, atribua o resultado de cada expressão a uma variável local e, então, passe essas variáveis como argumentos à função.*

O protótipo de função-membro `maximum` (Figura 6.3, linha 17) indica que a função retorna um valor do tipo inteiro, que o nome da função é `maximum` e que ela requer três parâmetros do tipo inteiro para realizar sua tarefa. O cabeçalho de função `maximum` (Figura 6.4, linha 62) corresponde ao protótipo de função e indica que os nomes de parâmetro são `x`, `y` e `z`. Quando `maximum` é chamado (Figura 6.4, linha 58), o parâmetro `x` é inicializado com o valor do argumento `grade1`, o parâmetro `y` é inicializado com o valor do argumento `grade2` e o parâmetro `z` é inicializado com o valor do argumento `grade3`. Deve haver um argumento na chamada de função para cada parâmetro (também chamado **parâmetro formal**) na definição de função.

Note que múltiplos parâmetros são especificados tanto no protótipo de função como no cabeçalho de função como uma lista separada por vírgulas. O compilador referencia o protótipo de função para verificar se as chamadas a `maximum` contêm o número e os tipos dos argumentos corretos e se esses tipos estão na ordem correta. Além disso, o compilador utiliza o protótipo para assegurar que o valor retornado pela função pode ser utilizado corretamente na expressão que chamou a função (por exemplo, uma chamada de função que retorna `void` não pode ser utilizada como o lado direito de uma instrução de atribuição). Cada argumento deve ser consistente com o tipo do parâmetro correspondente. Por exemplo, um parâmetro do tipo `double` pode receber valores como 7,35, 22 ou -0,03456, mas não uma string como "hello". Se os argumentos passados para uma função não corresponderem aos tipos especificados no protótipo da função, o compilador tenta converter os argumentos nesses tipos. A Seção 6.5 discute essa conversão.



### Erro comum de programação 6.1

*Declarar parâmetros de função do mesmo tipo que `double x`, `y` em vez de `double x`, `double y` é um erro de sintaxe — é necessário um tipo explícito para cada parâmetro na lista de parâmetros.*



### Erro comum de programação 6.2

*Os erros de compilação ocorrem se o protótipo de função, o cabeçalho de função e as chamadas de função realmente não corresponderem em número, tipo e ordem de argumentos e parâmetros e no tipo de retorno.*



### Observação de engenharia de software 6.5

*Uma função com muitos parâmetros pode realizar tarefas demais. Considere dividir a função em funções menores que realizam as tarefas separadas. Se possível, limite o cabeçalho de função a uma linha.*

Para determinar o valor máximo (linhas 62–75 da Figura 6.4), iniciamos com a suposição de que o parâmetro `x` contém o valor maior, então a linha 64 da função `maximum` declara a variável local `maximumValue` e a inicializa com o valor do parâmetro `x`. Naturalmente, é possível que o parâmetro `y` ou `z` contenham o maior valor real, portanto devemos comparar cada um desses valores com `maximumValue`. A instrução `if` nas linhas 67–68 determina se `y` é maior que `maximumValue` e, se o for, atribui `y` a `maximumValue`. A instrução `if` nas linhas 71–72 determina se `z` é maior que `maximumValue` e, se for, atribui `z` a `maximumValue`. Nesse ponto, o maior dos três valores está em `maximumValue`, então a linha 74 retorna esse valor à chamada na linha 58. Quando o controle do programa retornar ao ponto no código em que `maximum` foi chamado, os parâmetros `x`, `y` e `z` de `maximum` não serão mais acessíveis ao programa. Veremos por que na próxima seção.

Há três maneiras de retornar o controle para o ponto em que uma função foi invocada. Se a função não retornar um resultado (isto é, a função tem um tipo de retorno `void`), o controle retorna quando o programa alcançar a chave de fechamento direita da função ou pela execução da instrução

```
return;
```

Se a função retorna um resultado, a instrução

```
return expressão;
```

avalia *expressão* e retorna o valor de *expressão* ao chamador.

## 6.5 Protótipos de funções e coerção de argumentos

Um protótipo de função (também chamado de **declaração de função**) informa ao compilador o nome de uma função, o tipo de dados retornado pela função, o número de parâmetros que a função espera receber, os tipos desses parâmetros e a ordem em que eles são esperados.



### Observação de engenharia de software 6.6

*Os protótipos de função são requeridos em C++. Utilize as diretivas de pré-processador `#include` para obter protótipos de função das funções da C++ Standard Library a partir dos arquivos de cabeçalho para as bibliotecas apropriadas (por exemplo, o protótipo para a função de matemática `sqrt` está no arquivo de cabeçalho `<cmath>`; uma lista parcial de arquivos de cabeçalho C++ Standard Library aparece na Seção 6.6). Além disso, utilize `#include` para obter arquivos de cabeçalho que contêm protótipos de função escritos por você ou membros do seu grupo.*



### Erro comum de programação 6.3

*Se uma função for definida antes de ser invocada, então a definição da função também serve como o protótipo da função, portanto um protótipo separado é desnecessário. Se uma função é invocada antes de ser definida e essa função não tiver um protótipo de função, ocorre um erro de compilação.*



### Observação de engenharia de software 6.7

*Forneça sempre protótipos de função, mesmo que seja possível omiti-los quando as funções são definidas antes de serem utilizadas (caso em que o cabeçalho de função também atua como o protótipo de função). Fornecer os protótipos evita associar o código à ordem em que as funções são definidas (o que pode mudar facilmente à medida que o programa cresce).*

### Assinaturas de função

A parte de um protótipo de função que inclui o nome da função e os tipos de seus argumentos é chamada de **assinatura de função** ou simplesmente **assinatura**. A assinatura de função não especifica o tipo de retorno da função. A função no mesmo escopo deve ter assinaturas únicas. O escopo de uma função é a região de um programa em que a função é conhecida e acessível. Falaremos mais sobre escopo na Seção 6.10.



### Erro comum de programação 6.4

*É um erro de compilação se duas funções do mesmo escopo tiverem a mesma assinatura, mas diferentes tipos de retorno.*

Na Figura 6.3, se o protótipo de função na linha 17 fosse escrito

```
void maximum(int, int, int);
```

o compilador informaria um erro, porque o tipo de retorno `void` no protótipo de função iria diferir do tipo de retorno `int` no cabeçalho de função. De maneira semelhante, esse protótipo faria com que a instrução

```
cout << maximum(6, 9, 0);
```

gerasse um erro de compilação, porque essa instrução depende de `maximum` para retornar um valor a ser exibido.

### Coerção de argumento

Um recurso importante de protótipos de função é a **coerção de argumento** — isto é, forçar argumentos aos tipos apropriados especificados pelas declarações de parâmetro. Por exemplo, um programa pode chamar uma função com um argumento do tipo inteiro, mesmo que o protótipo de função especificar um argumento `double` — e a função ainda funcionará corretamente.

### Regras de promoção de argumento

Às vezes, os valores de argumento que não correspondem precisamente aos tipos de parâmetro no protótipo de função podem ser convertidos pelo compilador no tipo adequado antes que a função seja chamada. Essas conversões ocorrem de acordo com as especificações das **regras de promoção** do C++. As regras de promoção indicam como converter entre tipos sem perder dados. Um `int` pode ser convertido em um `double` sem alterar seu valor. Entretanto, um `double` convertido em um `int` trunca a parte fracionária do valor `double`. Tenha em mente que as variáveis `double` podem armazenar números de maior magnitude que as variáveis `int`, então a perda de dados pode ser considerável. Os valores também podem ser modificados ao converter tipos inteiro grandes em tipos inteiro pequenos (por exemplo, `long` em `short`), com sinal em sem sinal, ou sem sinal em com sinal.

As regras de promoção se aplicam a expressões que contêm valores de dois ou mais tipos de dados; essas expressões são também referidas como **expressões de tipo misto**. O tipo de cada valor em uma expressão de tipo misto é promovido para ‘o mais alto’ tipo na expressão (na realidade, uma versão temporária de cada valor é criada e utilizada para a expressão — os valores originais permanecem inalterados). A promoção também ocorre quando o tipo de um argumento de função não corresponde ao tipo de parâmetro especificado na definição ou protótipo de função. A Figura 6.6 lista os tipos de dados fundamentais na ordem do ‘tipo mais alto’ ao ‘tipo mais baixo’.

| Tipos de dados                                                             |
|----------------------------------------------------------------------------|
| <code>long double</code>                                                   |
| <code>double</code>                                                        |
| <code>float</code>                                                         |
| <code>unsigned long int</code> (sinônimo de <code>unsigned long</code> )   |
| <code>long int</code> (sinônimo de <code>long</code> )                     |
| <code>unsigned int</code> (sinônimo de <code>unsigned</code> )             |
| <code>int</code>                                                           |
| <code>unsigned short int</code> (sinônimo de <code>unsigned short</code> ) |
| <code>short int</code> (sinônimo de <code>short</code> )                   |
| <code>unsigned char</code>                                                 |
| <code>char</code>                                                          |
| <code>bool</code>                                                          |

**Figura 6.6** Hierarquia de promoção para tipos de dados fundamentais.

Converter valores em tipos fundamentais mais baixos pode resultar em valores incorretos. Portanto, um valor pode ser convertido em um tipo fundamental mais baixo apenas atribuindo explicitamente o valor a uma variável de tipo mais baixo (alguns compiladores emitirão um aviso nesse caso) ou utilizando um operador de coerção (ver Seção 4.9). Os valores de argumento de função são convertidos nos tipos de parâmetro em um protótipo de função como se estivessem sendo atribuídos diretamente às variáveis desses tipos. Se uma função `square` que utiliza um parâmetro de inteiro é chamada com um argumento de ponto flutuante, o argumento é convertido em `int` (um tipo mais baixo) e `square` poderia retornar um valor incorreto. Por exemplo, `square( 4.5 )` retorna 16, não 20.25.



### Erro comum de programação 6.5

*Converter de um tipo de dados mais alto, na hierarquia de promoção, em um tipo mais baixo, ou entre com sinal e sem sinal, pode corromper o valor dos dados, causando perda de informações.*



### Erro comum de programação 6.6

*É um erro de compilação se os argumentos em uma chamada de função não correspondem ao número e tipos dos parâmetros declarados no protótipo de função correspondente. Também é um erro se o número de argumentos na chamada for correspondido, mas os argumentos não puderem ser implicitamente convertidos nos tipos esperados.*

## 6.6 Arquivos de cabeçalho da biblioteca-padrão C++

A C++ Standard Library é dividida em muitas partes, cada qual com seu próprio arquivo de cabeçalho. Os arquivos de cabeçalho contêm os protótipos de função para as funções relacionadas que formam cada parte da biblioteca. Os arquivos de cabeçalho também contêm definições de vários tipos de classe e funções, bem como as constantes de que essas funções precisam. Um arquivo de cabeçalho ‘instrui’ o compilador a interfacear com a biblioteca e os componentes escritos pelo usuário.

A Figura 6.7 lista alguns arquivos de cabeçalho comuns da C++ Standard Library, a maioria dos quais é discutida mais adiante no livro. O termo ‘macro’ utilizado várias vezes na Figura 6.7 é discutido em detalhes no Apêndice F, “Pré-processador”. Os nomes de arquivo de cabeçalho que terminam em `.h` são arquivos de cabeçalho ‘no estilo antigo’ que foram substituídos pelos arquivos de cabeçalho da C++ Standard Library. Utilizamos neste livro apenas as versões da C++ Standard Library de cada arquivo de cabeçalho para assegurar que nossos exemplos funcionarão na maioria dos compiladores C++ padrão.

## 6.7 Estudo de caso: geração de números aleatórios

Agora vamos para uma breve e, esperamos, interessante diversão em um aplicativo de programação popular, a saber: a simulação e a execução de um jogo. Nesta e na próxima seção, desenvolvemos um programa de jogo que inclui múltiplas funções. O programa utiliza muitas das instruções de controle e conceitos discutidos até agora.

| Arquivo de cabeçalho<br>C++ Standard Library                                                                                                                                                                                                | Explicação                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>&lt;iostream&gt;</code>                                                                                                                                                                                                               | Contém protótipos de função para as funções de entrada e saída padrão do C++, introduzidas no Capítulo 2 e discutidas em detalhes no Capítulo 15, “Entrada/saída de fluxo”. Esse arquivo de cabeçalho substitui o arquivo de cabeçalho <code>&lt;iostream.h&gt;</code> .                                                                                                                                                                                                                                                                                                                |
| <code>&lt;iomanip&gt;</code>                                                                                                                                                                                                                | Contém protótipos de função para manipuladores de fluxo que formatam fluxos de dados. Esse arquivo de cabeçalho é inicialmente utilizado na Seção 4.9 e discutido em mais detalhes no Capítulo 15. Esse arquivo de cabeçalho substitui o arquivo de cabeçalho <code>&lt;iomanip.h&gt;</code> .                                                                                                                                                                                                                                                                                          |
| <code>&lt;cmath&gt;</code>                                                                                                                                                                                                                  | Contém protótipos de função para funções da biblioteca de matemática (discutidas na Seção 6.3). Esse arquivo de cabeçalho substitui o arquivo de cabeçalho <code>&lt;math.h&gt;</code> .                                                                                                                                                                                                                                                                                                                                                                                                |
| <code>&lt;cstdlib&gt;</code>                                                                                                                                                                                                                | Contém protótipos de função para conversões de números em texto, de texto em números, alocação de memória, números aleatórios e várias outras funções utilitárias. Partes do arquivo de cabeçalho são abrangidas na Seção 6.7; no Capítulo 11, “Sobrecarga de operadores; objetos string e array”; no Capítulo 16, “Tratamento de exceções”; no Capítulo 19, “Programação Web”; no Capítulo 22, “Bits, caracteres, strings C e structs”; e no Apêndice E, “Tópicos sobre o código C legado”. Esse arquivo de cabeçalho substitui o arquivo de cabeçalho <code>&lt;stdlib.h&gt;</code> . |
| <code>&lt;ctime&gt;</code>                                                                                                                                                                                                                  | Contém protótipos de função e tipos para manipular a data e a hora. Esse arquivo de cabeçalho substitui o arquivo de cabeçalho <code>&lt;time.h&gt;</code> . Esse arquivo de cabeçalho é utilizado na Seção 6.7.                                                                                                                                                                                                                                                                                                                                                                        |
| <code>&lt;vector&gt;</code> , <code>&lt;list&gt;</code> ,<br><code>&lt;deque&gt;</code> , <code>&lt;queue&gt;</code> ,<br><code>&lt;stack&gt;</code> , <code>&lt;map&gt;</code> ,<br><code>&lt;set&gt;</code> , <code>&lt;bitset&gt;</code> | Esses arquivos de cabeçalho contêm classes que implementam os contêineres da C++ Standard Library. Os contêineres armazenam dados durante a execução de um programa. O cabeçalho <code>&lt;vector&gt;</code> é introduzido no Capítulo 7, “Arrays e vetores”. Discutimos todos esses arquivos de cabeçalho no Capítulo 23, “Standard Template Library (STL)”.                                                                                                                                                                                                                           |
| <code>&lt;cctype&gt;</code>                                                                                                                                                                                                                 | Contém protótipos de função para funções que testam caracteres quanto a certas propriedades (como o fato de o caractere ser um dígito ou uma pontuação) e protótipos de função para funções que podem ser utilizadas para converter letras minúsculas em maiúsculas e vice-versa. Esse arquivo de cabeçalho substitui o arquivo de cabeçalho <code>&lt;ctype.h&gt;</code> . Esses tópicos são discutidos no Capítulo 8, “Ponteiros e strings baseadas em ponteiro” e no Capítulo 22, “Bits, caracteres, strings C e structs”.                                                           |
| <code>&lt;cstring&gt;</code>                                                                                                                                                                                                                | Contém protótipos de função para funções de processamento de string no estilo C. Esse arquivo de cabeçalho substitui o arquivo de cabeçalho <code>&lt;string.h&gt;</code> . Esse arquivo de cabeçalho é utilizado no Capítulo 11, “Sobrecarga de operadores; objetos string e array”.                                                                                                                                                                                                                                                                                                   |
| <code>&lt;typeinfo&gt;</code>                                                                                                                                                                                                               | Contém classes para identificação de tipo em tempo de execução (determinando tipos de dados em tempo de execução). Esse arquivo de cabeçalho é discutido na Seção 13.8.                                                                                                                                                                                                                                                                                                                                                                                                                 |
| <code>&lt;exception&gt;</code> ,<br><code>&lt;stdexcept&gt;</code>                                                                                                                                                                          | Esses arquivos de cabeçalho contêm classes que são utilizadas para tratamento de exceções (discutidos no Capítulo 16).                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
| <code>&lt;memory&gt;</code>                                                                                                                                                                                                                 | Contém classes e funções utilizadas pela C++ Standard Library para alocar memória aos contêineres da C++ Standard Library. Esse cabeçalho é utilizado no Capítulo 16, “Tratamento de exceções”.                                                                                                                                                                                                                                                                                                                                                                                         |
| <code>&lt;fstream&gt;</code>                                                                                                                                                                                                                | Contém protótipos de função para funções que realizam entrada de arquivos em disco e saída para arquivos em disco (discutido no Capítulo 17, “Processamento de arquivo”). Esse arquivo de cabeçalho substitui o arquivo de cabeçalho <code>&lt;fstream.h&gt;</code> .                                                                                                                                                                                                                                                                                                                   |
| <code>&lt;string&gt;</code>                                                                                                                                                                                                                 | Contém a definição de classe <code>string</code> da C++ Standard Library (discutida no Capítulo 18).                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <code>&lt;sstream&gt;</code>                                                                                                                                                                                                                | Contém protótipos de função para funções que realizam entrada de strings e saída para strings na memória (discutidos no Capítulo 18, “Classe <code>string</code> e processamento de fluxo de string”).                                                                                                                                                                                                                                                                                                                                                                                  |
| <code>&lt;functional&gt;</code>                                                                                                                                                                                                             | Contém classes e funções utilizadas por algoritmos da C++ Standard Library. Esse arquivo de cabeçalho é utilizado no Capítulo 23.                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| <code>&lt;iterator&gt;</code>                                                                                                                                                                                                               | Contém classes para acessar dados nos contêineres da C++ Standard Library. Esse arquivo de cabeçalho é utilizado no Capítulo 23, “Standard Template Library (STL)”.                                                                                                                                                                                                                                                                                                                                                                                                                     |
| <code>&lt;algorithm&gt;</code>                                                                                                                                                                                                              | Contém funções para manipular dados em contêineres da C++ Standard Library. Esse arquivo de cabeçalho é utilizado no Capítulo 23.                                                                                                                                                                                                                                                                                                                                                                                                                                                       |

Figura 6.7 Arquivos de cabeçalho da C++ Standard Library.

(continua)



| Arquivo de cabeçalho<br>C++ Standard Library | Explicação                                                                                                                                                                                                                              |
|----------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <cassert>                                    | Contém macros para adicionar diagnósticos que auxiliam na depuração de programa. Isso substitui o arquivo de cabeçalho <assert.h> do C++ pré-padrão. Esse arquivo de cabeçalho é utilizado no Apêndice F, “Pré-processador”.            |
| <cfloat>                                     | Contém os limites de tamanho de ponto flutuante do sistema. Esse arquivo de cabeçalho substitui o arquivo de cabeçalho <float.h>.                                                                                                       |
| <climits>                                    | Contém os limites de tamanho inteiros do sistema. Esse arquivo de cabeçalho substitui o arquivo de cabeçalho <limits.h>.                                                                                                                |
| <cstdio>                                     | Contém protótipos de função para as funções de biblioteca de entrada/saída padrão no estilo C e informações utilizadas por elas. Esse arquivo de cabeçalho substitui o arquivo de cabeçalho <stdio.h>.                                  |
| <locale>                                     | Contém classes e funções normalmente utilizadas pelo processamento de fluxo para processar dados na forma natural para diferentes idiomas (por exemplo, formatos monetários, classificação de strings, apresentação de caractere etc.). |
| <limits>                                     | Contém classes para definir os limites de tipos de dados numéricos em cada plataforma de computador.                                                                                                                                    |
| <utility>                                    | Contém classes e funções utilizadas por muitos arquivos de cabeçalho da C++ Standard Library.                                                                                                                                           |

Figura 6.7 Arquivos de cabeçalho da C++ Standard Library.

(continuação)

O elemento chance pode ser introduzido em aplicativos de computador utilizando a função `rand` da C++ Standard Library. Considere a seguinte instrução:

```
i = rand();
```

A função `rand` gera um inteiro sem sinal entre 0 e `RAND_MAX` (uma constante simbólica definida no arquivo de cabeçalho `<cstdlib>`). O valor de `RAND_MAX` deve ser pelo menos 32.767 — o valor positivo máximo de um inteiro de dois bytes (16 bits). Para o GNU C++, o valor de `RAND_MAX` é 214.748.647; para o Visual Studio, o valor de `RAND_MAX` é 32.767. Se `rand` produz verdadeiramente inteiros de modo aleatório, todo número entre 0 e `RAND_MAX` tem uma igual *chance* (ou **probabilidade**) de ser escolhido toda vez que `rand` é chamado.

O intervalo de valores produzido diretamente pela função `rand` é, com frequência, diferente daquele que um aplicativo específico requer. Por exemplo, um programa que simula lançamento de moeda talvez requeira somente 0 para ‘caras’ e 1 para ‘coroas’. Um programa que simula a rolagem de um dado de seis lados requer inteiros aleatórios no intervalo de 1 a 6. Um programa que aleatoriamente prevê o próximo tipo de nave espacial (uma entre quatro possibilidades) que voará no horizonte de um videogame poderia exigir inteiros aleatórios no intervalo de 1 a 4.

### Lançando um dado de seis faces

Para demonstrar `rand`, vamos desenvolver um programa (Figura 6.8) para simular 20 lançamentos de um dado de seis lados e imprimir o valor de cada lançamento. O protótipo de função para a função `rand` está em `<cstdlib>`. Para produzir inteiros no intervalo de 0 a 5, utilizamos o operador módulo (%) com `rand` como mostrado a seguir:

```
rand() % 6
```

Isso é chamado **escalonamento**. O número 6 é chamado de **fator de escalonamento**. Em seguida, **deslocamos** o intervalo de números produzidos adicionando 1 ao nosso resultado anterior. A Figura 6.8 confirma que os resultados estão no intervalo 1 a 6.

### Lançando um dado de seis faces 6.000.000 vezes

Para mostrar que os números produzidos pela função `rand` ocorrem aproximadamente com igual probabilidade, a Figura 6.9 simula 6.000.000 lançamentos de um dado. Cada inteiro no intervalo de 1 a 6 deve aparecer cerca de 1.000.000 vezes. Isso é confirmado pela janela de saída no fim da Figura 6.9.

Como mostra a saída de programa, você pode simular o lançamento de um dado de seis lados escalonando e deslocando os valores produzidos por `rand`. Observe que o programa nunca deve chegar ao caso `default` (linhas 50–51) fornecido na estrutura `switch`, porque a expressão de controle do `switch` (face) tem sempre valores no intervalo 1–6; entretanto, fornecemos o caso `default` por uma questão de boa prática. Depois de estudarmos os arrays no Capítulo 7, mostramos como substituir a estrutura `switch` inteira na Figura 6.9 elegantemente com uma instrução de uma única linha.



### Dica de prevenção de erro 6.3

Forneça um caso `default` em um `switch` para capturar erros mesmo se você estiver absolutamente certo de que não tem nenhum bug!

```

1 // Figura 6.8: fig06_08.cpp
2 // Inteiros aleatórios deslocados e escalonados.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 #include <cstdlib> // contém o protótipo de função para rand
11 using std::rand;
12
13 int main()
14 {
15 // itera 20 vezes
16 for (int counter = 1; counter <= 20; counter++)
17 {
18 // escolhe um número aleatório de 1 a 6 e o envia para saída
19 cout << setw(10) << (1 + rand() % 6);
20
21 // se o contador for divisível por 5, inicia uma nova linha de saída
22 if (counter % 5 == 0)
23 cout << endl;
24 } // fim do for
25
26 return 0; // indica terminação bem-sucedida
27 } // fim de main

```

|   |   |   |   |   |
|---|---|---|---|---|
| 6 | 6 | 5 | 5 | 6 |
| 5 | 1 | 1 | 5 | 3 |
| 6 | 6 | 2 | 4 | 2 |
| 6 | 2 | 3 | 4 | 1 |

**Figura 6.8** Inteiros deslocados e escalonados produzidos por  $1 + \text{rand}() \% 6$ .

```

1 // Figura 6.9: fig06_09.cpp
2 // Lança um dado de seis lados 6.000.000 vezes.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 #include <cstdlib> // contém o protótipo de função para rand
11 using std::rand;
12
13 int main()
14 {
15 int frequency1 = 0; // contagem de 1s lançados
16 int frequency2 = 0; // contagem de 2s lançados
17 int frequency3 = 0; // contagem de 3s lançados
18 int frequency4 = 0; // contagem de 4s lançados
19 int frequency5 = 0; // contagem de 5s lançados

```

**Figura 6.9** Lançando um dado de seis lados 6.000.000 vezes.

(continua)

```

20 int frequency6 = 0; // contagem de 6s lançados
21
22 int face; // armazena o valor lançado mais recentemente
23
24 // resume os resultados de 6,000,000 lançamentos de um dado
25 for (int roll = 1; roll <= 6000000; roll++)
26 {
27 face = 1 + rand() % 6; // número aleatório de 1 a 6
28
29 // determina valor de lançamento de 1 a 6 e incrementa o contador apropriado
30 switch (face)
31 {
32 case 1:
33 ++frequency1; // incrementa o contador de 1s
34 break;
35 case 2:
36 ++frequency2; // incrementa o contador de 2s
37 break;
38 case 3:
39 ++frequency3; // incrementa o contador de 3s
40 break;
41 case 4:
42 ++frequency4; // incrementa o contador de 4s
43 break;
44 case 5:
45 ++frequency5; // incrementa o contador de 5s
46 break;
47 case 6:
48 ++frequency6; // incrementa o contador de 6s
49 break;
50 default: // valor inválido
51 cout << "Program should never get here!";
52 } // fim de switch
53 } // fim do for
54
55 cout << "Face" << setw(13) << "Frequency" << endl; // cabeçalhos de saída
56 cout << " 1" << setw(13) << frequency1
57 << "\n 2" << setw(13) << frequency2
58 << "\n 3" << setw(13) << frequency3
59 << "\n 4" << setw(13) << frequency4
60 << "\n 5" << setw(13) << frequency5
61 << "\n 6" << setw(13) << frequency6 << endl;
62 return 0; // indica terminação bem-sucedida
63 } // fim de main

```

| Face | Frequency |
|------|-----------|
| 1    | 999702    |
| 2    | 1000823   |
| 3    | 999378    |
| 4    | 998898    |
| 5    | 1000777   |
| 6    | 1000422   |

Figura 6.9 Lançando um dado de seis lados 6.000.000 vezes.

(continuação)

*Aleatorizando o gerador de número aleatório*

Executar o programa da Figura 6.8 novamente produz

|   |   |   |   |   |
|---|---|---|---|---|
| 6 | 6 | 5 | 5 | 6 |
| 5 | 1 | 1 | 5 | 3 |
| 6 | 6 | 2 | 4 | 2 |
| 6 | 2 | 3 | 4 | 1 |

Note que o programa imprime exatamente a mesma sequência de valores mostrada na Figura 6.8. Como esses números podem ser aleatórios? Ironicamente, essa repetitividade é uma característica importante da função `rand`. Ao depurar um programa de simulação, essa repetitividade é essencial para comprovar que as correções no programa funcionam adequadamente.

A função `rand` na realidade gera **números pseudo-aleatórios** ou **pseudo-aleatórios**. Chamar `rand` repetidamente produz uma sequência de números que parece ser aleatória. Entretanto, a sequência repete-se toda vez que o programa executa. Uma vez que um programa foi completamente depurado, ele pode ser condicionado a produzir uma sequência diferente de números aleatórios para cada execução. Isso chama-se **aleatorização** ou **randomização** e é realizado com a função `srand` da C++ Standard Library. A função `srand` aceita um argumento do tipo inteiro `unsigned` e **semeia** a função `rand` para produzir uma sequência diferente de números aleatórios para cada execução do programa.

A Figura 6.10 demonstra a função `srand`. O programa utiliza o tipo de dados `unsigned`, que é a abreviação de `unsigned int`. Um `int` é armazenado pelo menos em dois bytes de memória (em geral, quatro bytes de memória nos sistemas de 32 bits populares de hoje) e pode ter valores positivo e negativo. Uma variável do tipo `unsigned int` também é armazenada em pelo menos dois bytes de memória. Um `unsigned int` de dois bytes pode ter apenas valores não negativos no intervalo 0–65535. Um `unsigned int` de quatro bytes pode ter somente valores não negativos no intervalo 0–4294967295. A função `srand` aceita um valor `unsigned int` como um argumento. O protótipo de função para a função `srand` está no arquivo de cabeçalho `<cstdlib>`.

Vamos executar o programa várias vezes e observar os resultados. Note que o programa produz uma sequência *diferente* de números aleatórios toda vez que ele executa, desde que o usuário insira uma semente diferente. Utilizamos a mesma semente nas primeira e terceira saídas de exemplo, então a mesma série de 10 números é exibida em cada uma dessas saídas.

Para aleatorizar, ou randomizar, sem inserir uma semente toda vez, podemos utilizar uma instrução como

```
srand(time(0));
```

Isso faz com que o computador leia o relógio dele para obter o valor para a semente. A função `time` (com o argumento 0 da maneira escrita na instrução precedente) retorna a hora atual como o número de segundos desde 1º de janeiro de 1970 à meia-noite do Greenwich Mean Time (GMT). Esse valor é convertido em um inteiro `unsigned` e utilizado como a semente para o gerador de número aleatório. O protótipo de função para `time` está em `<ctime>`.



### Erro comum de programação 6.7

Chamar a função `srand` mais de uma vez em um programa reinicia a sequência de números pseudo-aleatórios e pode afetar a aleatoriedade dos números produzidos por `rand`.

*Escalonamento e deslocamento generalizados de números aleatórios*

Anteriormente, demonstramos como escrever uma única instrução para simular a rolagem de um dado de seis lados com a instrução

```
face = 1 + rand() % 6;
```

que sempre atribui um inteiro (aleatoriamente) à variável `face` no intervalo  $1 \leq \text{face} \leq 6$ . Observe que a largura desse intervalo (isto é, o número de inteiros consecutivos no intervalo) é 6, e o número inicial no intervalo é 1. Examinando a instrução precedente, vemos que a largura do intervalo é determinada pelo número utilizado para escalonar `rand` com o operador módulo (isto é, 6), e o número inicial do intervalo é igual ao número (isto é, 1) que é adicionado à expressão `rand % 6`. Podemos generalizar esse resultado como

```
número = valorDeDeslocamento + rand() % fatorDeEscala;
```

onde **valorDeDeslocamento** é igual ao primeiro número no intervalo desejado de inteiros consecutivos e **fatorDeEscala** é igual à largura do intervalo desejado de inteiros consecutivos. Os exercícios mostram que é possível escolher inteiros aleatoriamente a partir de conjuntos de valores diferentes daqueles dos intervalos de inteiros consecutivos.



### Erro comum de programação 6.8

Utilizar `srand` no lugar de `rand` para tentar gerar números aleatórios é um erro de compilação — a função `srand` não retorna um valor.

```

1 // Figura 6.10: fig06_10.cpp
2 // Aleatorizando o programa de lançamento de dados.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 #include <iomanip>
9 using std::setw;
10
11 #include <cstdlib> // contém protótipos para funções srand e rand
12 using std::rand;
13 using std::srand;
14
15 int main()
16 {
17 unsigned seed; // armazena a semente inserida pelo usuário
18
19 cout << "Enter seed: ";
20 cin >> seed;
21 srand(seed); // semeia o gerador de número aleatório
22
23 // itera 10 vezes
24 for (int counter = 1; counter <= 10; counter++)
25 {
26 // escolhe um número aleatório de 1 a 6 e o envia para saída
27 cout << setw(10) << (1 + rand() % 6);
28
29 // se o contador for divisível por 5, inicia uma nova linha de saída
30 if (counter % 5 == 0)
31 cout << endl;
32 } // fim do for
33
34 return 0; // indica terminação bem-sucedida
35 } // fim de main

```

Enter seed: **67**

|   |   |   |   |   |
|---|---|---|---|---|
| 6 | 1 | 4 | 6 | 2 |
| 1 | 6 | 1 | 6 | 4 |

Enter seed: **432**

|   |   |   |   |   |
|---|---|---|---|---|
| 4 | 6 | 3 | 1 | 6 |
| 3 | 1 | 5 | 4 | 2 |

Enter seed: **67**

|   |   |   |   |   |
|---|---|---|---|---|
| 6 | 1 | 4 | 6 | 2 |
| 1 | 6 | 1 | 6 | 4 |

**Figura 6.10** Aleatorizando o programa de lançamento de dados.

## 6.8 Estudo de caso: jogo de azar e introdução a enum

Um dos jogos de azar mais populares é um jogo de dados conhecido como ‘*craps*’, que é jogado em cassinos e becos por todo o mundo. As regras do jogo são simples e diretas:

*Um jogador rola dois dados. Cada dado tem seis faces. Essas faces contêm 1, 2, 3, 4, 5 e 6 pontos. Depois que os dados param de rolar, a soma dos pontos nas faces viradas para cima é calculada. Se a soma é 7 ou 11 na primeira rolagem dos dados, o jogador ganha. Se a soma é 2, 3 ou 12 na primeira rolagem dos dados (chamado ‘craps’), o jogador perde (isto é, a ‘casa’ ganha). Se a soma*

*for 4, 5, 6, 8, 9 ou 10 na primeira rolagem dos dados, essa soma torna-se a 'pontuação' do jogador. Para ganhar, você deve continuar a lançar o dado até 'fazer sua pontuação'. O jogador perde se obtiver um 7 antes de fazer sua pontuação.*

O programa na Figura 6.11 simula o jogo de dados *craps*.

Nas regras do jogo, note que o jogador deve lançar dois dados na primeira e em todas as rolagens subsequentes. Definimos a função `rollDice` (linhas 71–83) para lançar o dado e calcular e imprimir sua soma. A função `rollDice` é definida uma vez, mas é chamada de dois lugares (linhas 27 e 51) no programa. Curiosamente, `rollDice` não aceita argumentos, então indicamos uma lista de parâmetros vazia no protótipo (linha 14) e no cabeçalho de função (linha 71). A função `rollDice` retorna a soma dos dois dados, então o tipo de retorno `int` é indicado no protótipo de função e no cabeçalho de função.

O jogo é razoavelmente complexo. O jogador pode ganhar ou perder no primeiro ou em qualquer lançamento subsequente. O programa utiliza a variável `gameStatus` para monitorar isso.

A variável `gameStatus` é declarada como do novo tipo `Status`. A linha 19 declara um tipo definido pelo usuário chamado **enumeração**. Uma enumeração, introduzida pela palavra-chave `enum` e seguida por um **nome de tipo** (nesse caso, `Status`), é um conjunto de constantes do tipo inteiro representadas por identificadores. Os valores dessas **constantes enumeradas** iniciam em 0, a menos que especificado de outro modo, e incrementam por 1.

Na enumeração anterior, a constante `CONTINUE` tem o valor 0, `WON` tem o valor 1 e `LOST` tem o valor 2. Os identificadores em uma `enum` devem ser únicos, mas as constantes enumeradas separadas podem ter o mesmo valor inteiro (em breve mostraremos como realizar isso).

```

1 // Figura 6.11: fig06_11.cpp
2 // Simulação do jogo de dados craps.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstdlib> // contém protótipos para funções srand e rand
8 using std::rand;
9 using std::srand;
10
11 #include <ctime> // contém protótipo para a função time
12 using std::time;
13
14 int rollDice(); // lança o dado, calcula e exibe a soma
15
16 int main()
17 {
18 // enumeração com constantes que representam o status do jogo
19 enum Status { CONTINUE, WON, LOST }; // todas as maiúsculas em constantes
20
21 int myPoint; // pontos se não ganhar ou perder na primeira rolagem
22 Status gameStatus; // pode conter CONTINUE, WON ou LOST
23
24 // torna aleatório o gerador de número aleatório utilizando a hora atual
25 srand(time(0));
26
27 int sumOfDice = rollDice(); // primeira rolagem dos dados
28
29 // determina status e pontuação do jogo (se necessário) com base no primeiro lançamento de
30 dados
31 switch (sumOfDice)
32 {
33 case 7: // ganha com 7 no primeiro lançamento
34 case 11: // ganha com 11 no primeiro lançamento
35 gameStatus = WON;
36 break;

```

**Figura 6.11** Simulação do jogo de dados *craps*.

(continua)



```

36 case 2: // perde com 2 no primeiro lançamento
37 case 3: // perde com 3 no primeiro lançamento
38 case 12: // perde com 12 no primeiro lançamento
39 gameStatus = LOST;
40 break;
41 default: // não ganhou nem perdeu, portanto registra a pontuação
42 gameStatus = CONTINUE; // jogo não terminou
43 myPoint = sumOfDice; // informa a pontuação
44 cout << "Point is " << myPoint << endl;
45 break; // opcional no final do switch
46 } // fim de switch
47
48 // enquanto o jogo não estiver completo
49 while (gameStatus == CONTINUE) // nem WON nem LOST
50 {
51 sumOfDice = rollDice(); // lança os dados novamente
52
53 // determina o status do jogo
54 if (sumOfDice == myPoint) // vitória por pontuação
55 gameStatus = WON;
56 else
57 if (sumOfDice == 7) // perde obtendo 7 antes de atingir a pontuação
58 gameStatus = LOST;
59 } // fim do while
60
61 // exibe uma mensagem ganhou ou perdeu
62 if (gameStatus == WON)
63 cout << "Player wins" << endl;
64 else
65 cout << "Player loses" << endl;
66
67 return 0; // indica terminação bem-sucedida
68 } // fim de main
69
70 // lança os dados, calcula a soma e exibe os resultados
71 int rollDice()
72 {
73 // seleciona valores aleatórios do dado
74 int die1 = 1 + rand() % 6; // primeiro lançamento do dado
75 int die2 = 1 + rand() % 6; // segundo lançamento do dado
76
77 int sum = die1 + die2; // calcula a soma de valores do dado
78
79 // exibe os resultados desse lançamento
80 cout << "Player rolled " << die1 << " + " << die2
81 << " = " << sum << endl;
82 return sum; // fim da função rollDice
83 } // fim da função rollDice

```

Player rolled 2 + 5 = 7  
Player wins

Player rolled 6 + 6 = 12  
Player loses

Figura 6.11 Simulação do jogo de dados craps.

(continua)

```

Player rolled 3 + 3 = 6
Point is 6
Player rolled 5 + 3 = 8
Player rolled 4 + 5 = 9
Player rolled 2 + 1 = 3
Player rolled 1 + 5 = 6
Player wins

```

```

Player rolled 1 + 3 = 4
Point is 4
Player rolled 4 + 6 = 10
Player rolled 2 + 4 = 6
Player rolled 6 + 4 = 10
Player rolled 2 + 3 = 5
Player rolled 2 + 4 = 6
Player rolled 1 + 1 = 2
Player rolled 4 + 4 = 8
Player rolled 4 + 3 = 7
Player loses

```

**Figura 6.11** Simulação do jogo de dados *craps*.

(continuação)



### Boa prática de programação 6.1

*Torne maiúscula a primeira letra de um identificador utilizado como um nome de tipo definido pelo usuário.*



### Boa prática de programação 6.2

*Utilize somente letras maiúsculas nos nomes das constantes enumeradas. Isso faz com que essas constantes sejam destacadas em um programa e lembra o programador de que constantes enumeradas não são variáveis.*

As variáveis do tipo definido pelo usuário `Status` podem receber somente um dos três valores declarados na enumeração. Quando o jogo é ganho, o programa configura a variável `gameStatus` como `WON` (linhas 34 e 55). Quando o jogo é perdido, o programa configura a variável `gameStatus` como `LOST` (linhas 39 e 58). Caso contrário, o programa configura a variável `gameStatus` como `CONTINUE` (linha 42) para indicar que os dados devem ser rolados novamente.

Outra enumeração popular é

```

enum Months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG,
 SEP, OCT, NOV, DEC };

```

que cria o tipo definido pelo usuário `Months` com constantes enumeradas que representam os meses do ano. O primeiro valor na enumeração anterior é explicitamente configurado como 1, desse modo os valores restantes incrementam a partir de 1, resultando nos valores 1 a 12. Qualquer enumeração constante pode receber um valor inteiro na definição de enumeração, e cada uma das constantes enumeradas subsequentes tem um valor 1 mais alto que a constante anterior na lista até a próxima configuração explícita.

Depois do primeiro lançamento, se o jogo é ganho ou perdido, o programa pula o corpo da instrução `while` (linhas 49–59) porque `gameStatus` não é igual a `CONTINUE`. O programa prossegue com a instrução `if...else` nas linhas 62–65, o que imprime "Player wins" se `gameStatus` for igual a `WON` e "Player loses" se `gameStatus` for igual a `LOST`.

Depois do primeiro lançamento, se o jogo não acabar, o programa salva a soma em `myPoint` (linha 43). A execução prossegue com a instrução `while`, porque `gameStatus` é igual a `CONTINUE`. Durante cada iteração do `while`, o programa chama `rollDice` para produzir uma nova soma. Se a soma corresponder a `myPoint`, o programa configura `gameStatus` como `WON` (linha 55), o teste de `while` falha, a instrução `if...else` imprime "Player wins" e a execução termina. Se a soma for igual a 7, o programa configura `gameStatus` como `LOST` (linha 58), o teste de `while` falha, a instrução `if...else` imprime "Player loses" e a execução termina.

Observe a interessante utilização dos vários mecanismos de controle de programa que já discutimos. O programa do jogo de dados *craps* utiliza duas funções — `main` e `rollDice` — e as instruções `switch`, `while`, `if...else`, `if...else` aninhadas e `if` aninhadas. Nos exercícios, investigamos várias características interessantes da execução do jogo de dados.



### Boa prática de programação 6.3

*Utilizar enumerações em vez de constantes do tipo inteiro pode tornar os programas mais claros e mais sustentáveis. Você pode configurar o valor de uma constante enumerada uma vez, na declaração da enumeração.*

**Erro comum de programação 6.9**

*Atribuir o equivalente inteiro de uma constante enumerada a uma variável do tipo enumerado é um erro de compilação.*

**Erro comum de programação 6.10**

*Depois que uma constante enumerada tiver sido definida, tentar atribuir outro valor à constante enumerada é um erro de compilação.*

**6.9 Classes de armazenamento**

Os programas que você viu até agora utilizam identificadores para nomes de variáveis. Os atributos de variáveis incluem nome, tipo, tamanho e valor. Este capítulo também usa identificadores como nomes para funções definidas pelo usuário. De fato, todo identificador em um programa tem outros atributos, incluindo **classe de armazenamento**, escopo e **linkagem**.

O C++ fornece cinco **especificadores de classe de armazenamento**: **auto**, **register**, **extern**, **mutable** e **static**. Esta seção discute os especificadores de classe de armazenamento **auto**, **register**, **extern** e **static**. O especificador de classe de armazenamento **mutable** (discutido em detalhes no Capítulo 24) é utilizado exclusivamente com classes.

*Classe de armazenamento, escopo e linkagem*

A classe de armazenamento de um identificador determina o período em que esse identificador existe na memória. Alguns identificadores existem brevemente, algum são criados repetidamente e destruídos, e outros existem por toda a execução de um programa. Esta seção discute duas classes de armazenamento: **estática** e **automática**.

O escopo de um identificador é onde o identificador pode ser referenciado em um programa. Alguns identificadores podem ser referenciados por todo um programa; outros podem ser referenciados somente a partir de áreas limitadas de um programa. A Seção 6.10 discute o escopo dos identificadores.

A linkagem de um identificador determina se um identificador só é conhecido no arquivo de fonte onde ele é declarado ou por múltiplos arquivos que são compilados e, então, linkados. O especificador de classe de armazenamento de um identificador ajuda a determinar sua classe de armazenamento e linkagem.

*Categorias de classe de armazenamento*

Os especificadores de classe de armazenamento podem ser divididos em duas classes de armazenamento: classe de armazenamento automática e classe de armazenamento estática. As palavras-chave **auto** e **register** são utilizadas para declarar as variáveis da classe de armazenamento automática. Essas variáveis são criadas quando a execução do programa insere o bloco em que elas são definidas, existem enquanto o bloco está ativo e são destruídas quando o programa sai do bloco.

*Variáveis locais*

Somente variáveis locais de uma função podem ser de classe de armazenamento automática. As variáveis locais e parâmetros de uma função normalmente são da classe de armazenamento automática. O especificador de classe de armazenamento **auto** declara explicitamente variáveis de classe de armazenamento automática. Por exemplo, a seguinte declaração indica que as variáveis `double x` e `y` são variáveis locais da classe de armazenamento automática — elas só existem no par de chaves de fechamento mais próximo dentro do corpo da função em que a definição aparece:

```
auto double x, y;
```

As variáveis locais são da classe de armazenamento automática por padrão, então a palavra-chave **auto** raramente é utilizada. Para o restante do texto, iremos nos referir às variáveis de classe de armazenamento automática simplesmente como variáveis automáticas.

**Dica de desempenho 6.1**

*O armazenamento automático é um meio de economizar memória, porque as variáveis de classe de armazenamento automática só existem na memória quando o bloco em que são definidas estiver executando.*

**Observação de engenharia de software 6.8**

*O armazenamento automático é um exemplo do **princípio do menor privilégio**, que é fundamental para a boa engenharia de software. No contexto de um aplicativo, o princípio declara que deve ser concedida ao código somente a quantidade de privilégio e acesso de que ele precisa para realizar sua tarefa designada, não mais que isso. Por que devemos ter variáveis armazenadas na memória e acessíveis quando não são necessárias?*

*Variáveis de registro*

Os dados na versão de linguagem de máquina de um programa normalmente são carregados em registros para cálculos e outros processamentos.



### Dica de desempenho 6.2

*O especificador de classe de armazenamento `register` pode ser colocado antes de uma declaração de variável automática para sugerir que o compilador mantém a variável em um dos registros de hardware de alta velocidade do computador em vez de na memória. Se variáveis intensamente utilizadas como contadores ou totais são mantidas em registros de hardware, elimina-se o overhead de carregar repetidamente as variáveis de memória nos registros e armazenar os resultados de volta na memória.*



### Erro comum de programação 6.11

*Utilizar múltiplos especificadores de classe de armazenamento para um identificador é um erro de sintaxe. Somente um especificador de classe de armazenamento pode ser aplicado a um identificador. Por exemplo, se você incluir `register`, não inclua também `auto`.*

O compilador talvez ignore declarações `register`. Por exemplo, talvez não haja um número suficiente de registros disponíveis para o compilador utilizar. A seguinte definição *sugere* que a variável de inteiro `counter` seja colocada em um dos registros do computador; independentemente de o compilador fazer isso, `counter` é inicializado como 1:

```
register int counter = 1;
```

A palavra-chave `register` só pode ser utilizada com variáveis locais e parâmetros de função.



### Dica de desempenho 6.3

*Freqüentemente, a palavra-chave `register` é desnecessária. Os atuais compiladores de otimização são capazes de reconhecer freqüentemente variáveis utilizadas e podem decidir colocá-las em registros sem precisar de uma declaração `register` do programador.*

#### Classe de armazenamento estática

As palavras-chave `extern` e `static` declaram identificadores para variáveis da classe de armazenamento estática e para funções. As variáveis de classe de armazenamento estática existem a partir do ponto em que o programa começa a execução e duram até o fim do programa. O armazenamento de uma variável de uma classe de armazenamento estático é alocado quando o programa começa a execução. Essa variável é inicializada uma vez, quando sua declaração é encontrada. Para funções, o nome da função existe a partir do momento em que o programa começa a executar, assim como ocorre para todas as outras funções. Entretanto, mesmo que as variáveis e os nomes de função existam desde o início da execução do programa, isso não significa que esses identificadores podem ser utilizados por todo o programa. A classe de armazenamento e escopo (onde um nome pode ser utilizado) são questões separadas, como veremos na Seção 6.10.

#### Identificadores com classe de armazenamento estática

Há dois tipos de identificadores com classe de armazenamento estática — os identificadores externos (como as **variáveis globais** e os nomes de função globais) e as variáveis locais declaradas com o especificador de classe de armazenamento `static`. As variáveis globais são criadas colocando-se as declarações de variável fora de qualquer classe ou definição de função. As variáveis globais retêm seus valores por toda a execução do programa. Variáveis globais e funções globais podem ser referenciadas por qualquer função que siga suas declarações ou definições no arquivo-fonte.



### Observação de engenharia de software 6.9

*Declarar uma variável como global em vez de declará-la como local permite que ocorram efeitos colaterais indesejáveis quando uma função que não precisa de acesso à variável a modifica acidental ou maliciosamente. Esse é outro exemplo do princípio do menor privilégio. Em geral, exceto por recursos verdadeiramente globais como `cin` e `cout`, o uso de variáveis globais deve ser evitado a não ser em certas situações com requisitos de desempenho únicos.*



### Observação de engenharia de software 6.10

*As variáveis utilizadas apenas em uma função particular devem ser declaradas como variáveis locais nessa função, em vez de ser declaradas como variáveis globais.*

As variáveis locais declaradas com a palavra-chave `static` ainda são conhecidas apenas na função em que são declaradas, mas, ao contrário das variáveis automáticas, as variáveis locais `static` retêm seus valores quando a função retorna para seu chamador. A próxima vez que a função é chamada, as variáveis locais `static` contêm os valores que tinham quando a função completou pela última vez a execução. A instrução seguinte declara a variável local `count` como `static` e a inicializa como 1:

```
static int count = 1;
```

Todas as variáveis numéricas da classe de armazenamento estática são inicializadas como zero se não forem explicitamente inicializadas pelo programador, mas, de qualquer maneira, é uma boa prática inicializar explicitamente todas as variáveis.

Os especificadores de classe de armazenamento `extern` e `static` têm significado especial quando aplicados explicitamente a identificadores externos como variáveis globais e nomes de função globais. No Apêndice E, “Tópicos sobre o código C legado”, discutimos como utilizar `extern` e `static` com identificadores externos e programas de múltiplos arquivos-fonte.

## 6.10 Regras de escopo

A parte do programa em que um identificador pode ser utilizado é conhecida como escopo. Por exemplo, quando declaramos uma variável local em um bloco, ela pode ser referenciada apenas nesse bloco e nos blocos aninhados dentro desse bloco. Esta seção discute quatro escopos para um identificador — **escopo de função**, **escopo de arquivo**, **escopo de bloco** e **escopo de protótipo de função**. Mais adiante examinaremos dois outros escopos — **escopo de classe** (Capítulo 9) e **escopo de namespaces** (Capítulo 24).

Um identificador declarado fora de qualquer função ou classe tem escopo de arquivo. Tal identificador é ‘conhecido’ em todas as funções desde o ponto em que ele é declarado até o fim do arquivo. Todas as variáveis globais, definições de função e protótipos de função colocados fora de uma função têm escopo de arquivo.

Os **rótulos** (identificadores seguidos por dois-pontos como `start:`) são os únicos identificadores com escopo de função. Os rótulos podem ser utilizados em qualquer lugar na função em que aparecem, mas não podem ser referenciados fora do corpo da função. Os rótulos são utilizados em instruções `goto` (Apêndice E). Os rótulos são detalhes de implementação que as funções ocultam umas das outras.

Os identificadores declarados dentro de um bloco têm escopo de bloco. O escopo de bloco começa na declaração do identificador e termina na chave de fechamento direita `}` do bloco em que o identificador é declarado. As variáveis locais têm escopo de bloco, assim como os parâmetros de função, que também são variáveis locais da função. Todo bloco pode conter declarações de variável. Quando os blocos são aninhados e um identificador em um bloco externo tem o mesmo nome de um identificador em um bloco interno, o identificador no bloco externo fica ‘oculto’ até que o bloco interno termine. Durante a execução no bloco interno, este vê o valor de seu próprio identificador local e não o valor do identificador chamado identicamente no bloco que envolve. As variáveis locais declaradas `static` ainda têm escopo de bloco, mesmo que existam a partir do momento em que o programa começa a executar. A duração do armazenamento não afeta o escopo de um identificador.

Os únicos identificadores com escopo de protótipo de função são aqueles utilizados na lista de parâmetros de um protótipo de função. Como mencionado anteriormente, os protótipos de função não exigem nomes na lista de parâmetros — apenas os tipos são necessários. Os nomes que aparecem na lista de parâmetros de um protótipo de função são ignorados pelo compilador. Os identificadores utilizados em um protótipo de função podem ser reutilizados em outra parte no programa sem ambigüidade. Em um único protótipo, um identificador particular pode ser utilizado apenas uma vez.



### Erro comum de programação 6.12

*Normalmente é um erro de lógica utilizar acidentalmente o mesmo nome de um identificador em um bloco interno que é utilizado para um identificador em um bloco externo, quando de fato o programador quer que o identificador no bloco externo esteja ativo até o fim do bloco interno.*



### Boa prática de programação 6.4

*Evite nomes de variáveis que ocultam nomes em escopos externos. Isso pode ser realizado evitando-se o uso de identificadores duplicados em um programa.*

O programa da Figura 6.12 demonstra questões de escopo com variáveis globais, variáveis locais automáticas e variáveis locais `static`.

A linha 11 declara e inicializa a variável global `x` como 1. Essa variável global é ocultada em todo bloco (ou função) que declara uma variável chamada `x`. Em `main`, a linha 15 declara uma variável local `x` e a inicializa como 5. A linha 17 gera saída dessa variável para mostrar que `x` global está oculto em `main`. Em seguida, as linhas 19–23 definem um novo bloco em `main` em que outra variável local `x` é inicializada como 7 (linha 20). A linha 22 gera saída dessa variável para mostrar que ela oculta `x` no bloco externo de `main`. Quando o fluxo do programa sai do bloco, a variável `x` com o valor 7 é automaticamente destruída. Em seguida, a linha 25 gera saída da variável local `x` no bloco externo de `main` para mostrar que ela não está mais oculta.

Para demonstrar outros escopos, o programa define três funções, que não aceitam argumentos e não retornam nada. A função `useLocal` (linhas 39–46) declara a variável automática `x` (linha 41) e a inicializa como 25. Quando o programa chama `useLocal`, a função imprime a variável, incrementa essa variável e a imprime novamente antes de a função retornar o controle de programa ao seu chamador. Toda vez que o programa chamar essa função, ela recriará a variável automática `x` e a inicializará novamente como 25.

A função `useStaticLocal` (linhas 51–60) declara a variável `static x` e a inicializa como 50. As variáveis locais declaradas como `static` reêm seus valores mesmo quando estão fora de escopo (isto é, a função em que são declaradas não está em execução). Quando o programa chama `useStaticLocal`, a função imprime `x`, incrementa `x` e o imprime novamente antes de a função devolver o controle do programa ao seu chamador. Na próxima chamada a essa função, a variável local `static x` contém o valor 51. A inicialização na linha 53 só ocorre uma vez — na primeira vez em que `useStaticLocal` é chamada.

```

1 // Figura 6.12: fig06_12.cpp
2 // Um exemplo de escopo.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 void useLocal(void); // protótipo de função
8 void useStaticLocal(void); // protótipo de função
9 void useGlobal(void); // protótipo de função
10
11 int x = 1; // variável global
12
13 int main()
14 {
15 int x = 5; // variável local para main
16
17 cout << "local x in main's outer scope is " << x << endl;
18
19 { // inicia novo escopo
20 int x = 7; // oculta x no escopo externo
21
22 cout << "local x in main's inner scope is " << x << endl;
23 } // fim do novo escopo
24
25 cout << "local x in main's outer scope is " << x << endl;
26
27 useLocal(); // useLocal tem uma variável local x
28 useStaticLocal(); // useStaticLocal tem x estático local
29 useGlobal(); // useGlobal utiliza x global
30 useLocal(); // useLocal reinicializa seu x local
31 useStaticLocal(); // x estático local retém seu valor anterior
32 useGlobal(); // x global também retém seu valor
33
34 cout << "\nlocal x in main is " << x << endl;
35 return 0; // indica terminação bem-sucedida
36 } // fim de main
37
38 // useLocal reinicializa a variável local x durante cada chamada
39 void useLocal(void)
40 {
41 int x = 25; // inicializada toda vez que useLocal é chamada
42
43 cout << "\nlocal x is " << x << " on entering useLocal" << endl;
44 x++;
45 cout << "local x is " << x << " on exiting useLocal" << endl;
46 } // fim da função useLocal
47
48 // useStaticLocal inicializa a variável estática local x somente
49 // na primeira vez em que a função é chamada; o valor de x é salvo
50 // entre as chamadas a essa função
51 void useStaticLocal(void)
52 {
53 static int x = 50; // inicializada na primeira vez em que useStaticLocal é chamada
54
55 cout << "\nlocal static x is " << x << " on entering useStaticLocal"
56 << endl;
57 x++;

```

**Figura 6.12** Exemplo de escopo.

(continua)

```

58 cout << "local static x is " << x << " on exiting useStaticLocal"
59 << endl;
60 } // fim da função useStaticLocal
61
62 // useGlobal modifica a variável global x durante cada chamada
63 void useGlobal(void)
64 {
65 cout << "\nglobal x is " << x << " on entering useGlobal" << endl;
66 x *= 10;
67 cout << "global x is " << x << " on exiting useGlobal" << endl;
68 } // fim da função useGlobal

```

```

local x in main's outer scope is 5
local x in main's inner scope is 7
local x in main's outer scope is 5

local x is 25 on entering useLocal
local x is 26 on exiting useLocal

local static x is 50 on entering useStaticLocal
local static x is 51 on exiting useStaticLocal

global x is 1 on entering useGlobal
global x is 10 on exiting useGlobal

local x is 25 on entering useLocal
local x is 26 on exiting useLocal

local static x is 51 on entering useStaticLocal
local static x is 52 on exiting useStaticLocal

global x is 10 on entering useGlobal
global x is 100 on exiting useGlobal

local x in main is 5

```

Figura 6.12 Exemplo de escopo.

(continuação)

A função `useGlobal` (linhas 63–68) não declara nenhuma variável. Portanto, ao referenciar a variável `x`, o `x` global (precedendo `main`) é utilizado. Quando o programa chama `useGlobal`, a função imprime a variável global `x`, multiplica-a por 10 e a imprime novamente antes de a função devolver o controle do programa ao seu chamador. Na próxima vez em que o programa chama `useGlobal`, a variável global tem seu valor modificado, 10. Depois de executar as funções `useLocal`, `useStaticLocal` e `useGlobal` duas vezes cada, o programa imprime a variável local `x` em `main` novamente para mostrar que nenhuma das chamadas de função modificou o valor de `x` em `main`, porque todas as funções referenciavam as variáveis em outros escopos.

## 6.1.1 Pilha de chamadas de função e registros de ativação

Para entender como o C++ realiza chamadas de função, primeiro precisamos considerar uma estrutura de dados (isto é, uma coleção de itens de dados relacionados) conhecida como **pilha**. Pense em uma pilha de chamadas de função como algo análogo a uma pilha de pratos. Quando um prato é colocado na pilha, normalmente ele é colocado na parte superior (o que é conhecido como **inserir** o prato na pilha). De maneira semelhante, quando um prato é removido da pilha, normalmente ele é removido da parte superior (conhecido como **retirar** o prato da pilha). As pilhas são conhecidas como estruturas de dados do tipo **último a entrar, primeiro a sair** (*last-in, first-out* – **LIFO**) — o último item inserido na pilha é o primeiro item que é removido da pilha.

Um dos mecanismos mais importantes para os alunos de ciência da computação entenderem é o da **pilha de chamadas de função** (às vezes referida como a **pilha de execução do programa**). Essa estrutura de dados — que funciona ‘nos bastidores’ — suporta o mecanismo de chamada/retorno de função. Ela suporta também a criação, manutenção e destruição de variáveis automáticas de cada função chamada. Explicamos o comportamento de pilhas último a entrar, primeiro a sair (*last-in, first-out* – **LIFO**) com nosso exemplo de



empilhamento de pratos. Como veremos nas figuras 6.14–6.16, esse comportamento LIFO é exatamente o que uma função faz ao retornar à função que a chamou.

À medida que cada função é chamada, ela pode, por sua vez, chamar outras funções que, por sua vez, podem chamar outras funções — tudo antes mesmo de qualquer uma das funções retornar. Por fim, cada função deve devolver o controle à função que ela chamou. Portanto, de certo modo, devemos monitorar os endereços de retorno de que cada função precisa para retornar o controle à função que ela chamou. A pilha de chamadas de função é a estrutura de dados perfeita para o tratamento dessas informações. Toda vez que uma função chama outra função, uma entrada é empurrada sobre a pilha. Essa entrada, chamada de **quadro de pilha** ou **registro de ativação**, contém o endereço de retorno de que a função chamada precisa para retornar à função chamadora. Ela também contém algumas informações adicionais que logo discutiremos. Se a função chamada retorna, em vez de chamar outra função antes de retornar, o quadro de pilha (*stack frame*) para a chamada de função é removido e o controle é transferido para o endereço de retorno no quadro de pilha removido.

O apelo da pilha de chamadas é que cada função chamada sempre localiza as informações de que precisa para retornar ao seu chamador na parte superior da pilha de chamadas. E, se uma função faz uma chamada para outra função, um quadro de pilha para a nova chamada de função é simplesmente inserido na pilha de chamadas. Portanto, o endereço de retorno requerido pela função recém-chamada para retornar ao seu chamador agora está localizado na parte superior da pilha.

Os quadros de pilha têm outra responsabilidade importante. A maioria das funções tem variáveis automáticas — parâmetros e quaisquer variáveis locais que a função declarar. As variáveis automáticas precisam existir enquanto uma função estiver em execução. Elas precisam permanecer ativas se a função fizer chamadas para outras funções. Mas quando uma função chamada retorna à sua chamadora, as variáveis automáticas da função chamada precisam ‘desaparecer’. O quadro de pilha da função chamada é um lugar perfeito para reservar a memória para as variáveis automáticas da função chamada. Esse quadro de pilha existe enquanto a função chamada estiver ativa. Quando a função chamada retornar — e não precisar mais de suas variáveis automáticas locais —, seu quadro de pilha é removido da pilha e essas variáveis automáticas locais não são mais conhecidas pelo programa.

Naturalmente, a quantidade de memória em um computador é finita, portanto somente certa quantidade de memória pode ser utilizada para armazenar os registros de ativação na pilha de chamadas de função. Se houver mais chamadas de função do que as que podem ter os seus registros de ativação armazenados na pilha de chamadas de função, ocorre um erro conhecido como **estouro de pilha**.

### *Pilha de chamadas de função em ação*

Portanto, como vimos, a pilha de chamadas e registros de ativação suportam o mecanismo de chamada/retorno de função e a criação e destruição de variáveis automáticas. Agora consideremos a maneira como a pilha de chamadas suporta a operação de uma função `square` chamada por `main` (linhas 11–17 da Figura 6.13). Primeiro o sistema operacional chama `main` — isso insere um registro de ativação na

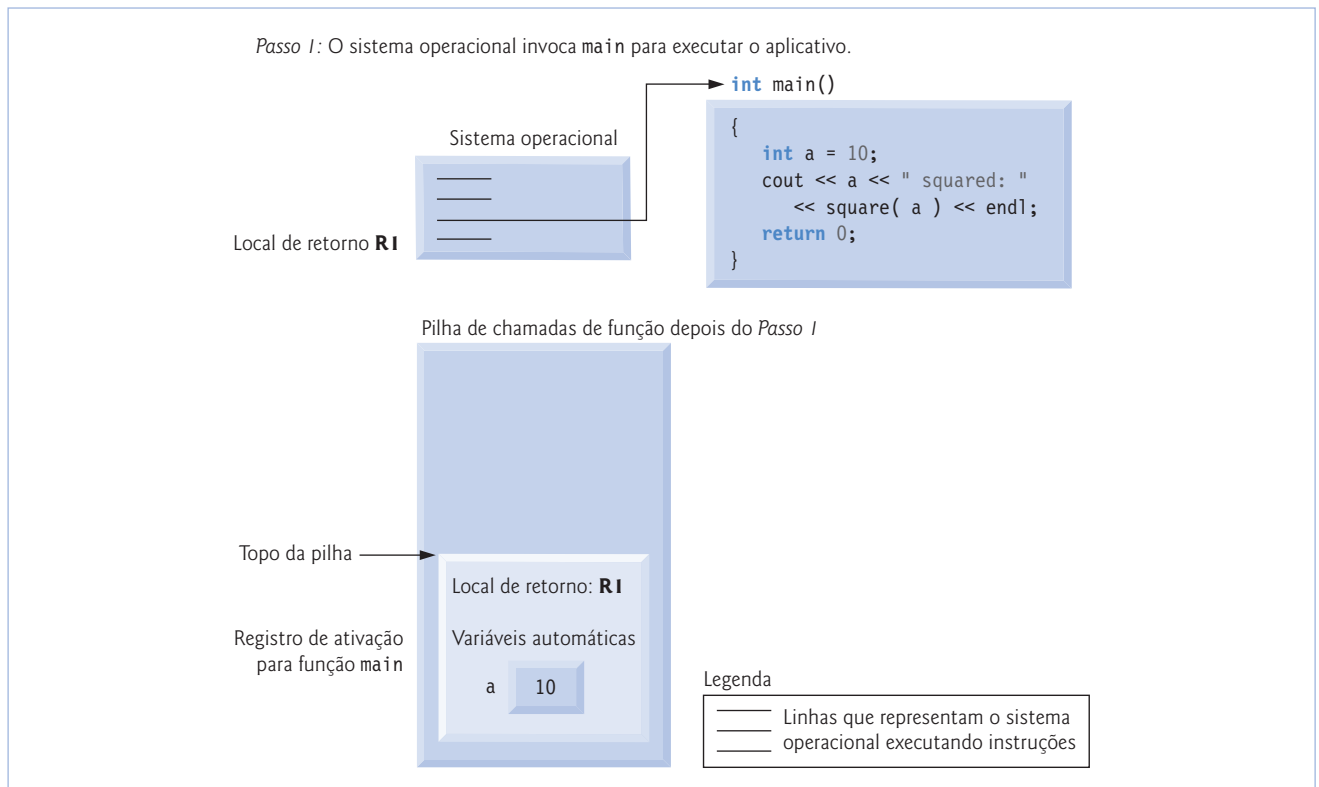
```

1 // Figura 6.13: fig06_13.cpp
2 // Função square utilizada para demonstrar a pilha
3 // de chamadas de função e os registros de ativação.
4 #include <iostream>
5 using std::cin;
6 using std::cout;
7 using std::endl;
8
9 int square(int); // protótipo para a função square
10
11 int main()
12 {
13 int a = 10; // valor para square (variável automática local em main)
14
15 cout << a << " squared: " << square(a) << endl; // exibe o quadrado de um int
16 return 0; // indica terminação bem-sucedida
17 } // fim de main
18
19 // retorna o quadrado de um inteiro
20 int square(int x) // x é uma variável local
21 {
22 return x * x; // calcula square e retorna o resultado
23 } // fim da função square

```

10 squared: 100

**Figura 6.13** A função `square` utilizada para demonstrar a pilha de chamadas de função e os registros de ativação.



**Figura 6.14** A pilha de chamadas de função depois que o sistema operacional invoca `main` para executar o aplicativo.

pilha (mostrado na Figura 6.14). O registro de ativação instrui `main` sobre como retornar ao sistema operacional (isto é, transferir para retornar o endereço `R1`) e contém o espaço para variável automática de `main` (isto é, `a`, que é inicializada como 10).

A função `main` — antes de retornar ao sistema operacional — agora chama a função `square` na linha 15 da Figura 6.13. Isso faz com que um quadro de pilha para `square` (linhas 20–23) seja inserido na pilha de chamadas de função (Figura 6.15). Esse quadro de pilha contém o endereço de retorno de que `square` precisa para retornar a `main` (isto é, `R2`) e a memória para a variável automática de `square` (isto é, `x`).

Depois de calcular o quadrado de seu argumento, `square` precisa retornar a `main` — e não precisa mais da memória para sua variável automática `x`. Portanto, o quadro de pilha da função é removido da pilha e `square` recebe a localização de retorno em `main` (isto é, `R2`) e perde sua variável automática. A Figura 6.16 mostra a pilha de chamadas de função depois de o registro de ativação de `square` ter sido removido.

A função `main` agora exibe o resultado da chamada a `square` (linha 15) e então executa a instrução `return` (linha 16). Isso faz com que o registro de ativação para `main` seja removido da pilha. Isso fornece o endereço necessário para `main` retornar ao sistema operacional (isto é, `R1` na Figura 6.14) e faz com que a memória da variável automática de `main` (isto é, `a`) se torne indisponível.

Você viu agora o quanto é valiosa a noção da estrutura de dados de pilha ao implementar um mecanismo-chave que suporta a execução do programa. As estruturas de dados têm muitas aplicações importantes na ciência da computação. Discutimos as pilhas, filas, listas, árvores e outras estruturas de dados no Capítulo 21, “Estruturas de dados”, e no Capítulo 23, “Standard Template Library (STL)”.

## 6.12 Funções com listas de parâmetro vazias

Em C++, uma lista de parâmetros vazia é especificada escrevendo-se `void` ou simplesmente nada entre parênteses. O protótipo

```
void print();
```

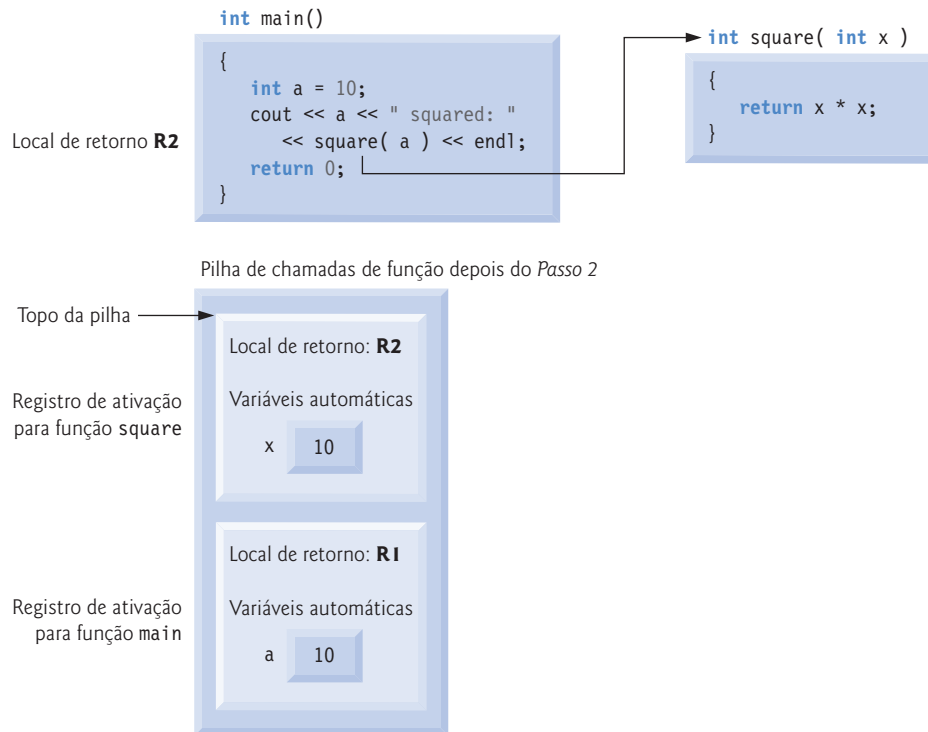
especifica que a função `print` não aceita argumentos e não retorna um valor. A Figura 6.17 demonstra as maneiras de declarar e utilizar funções com listas de parâmetros vazias.



### Dica de portabilidade 6.2

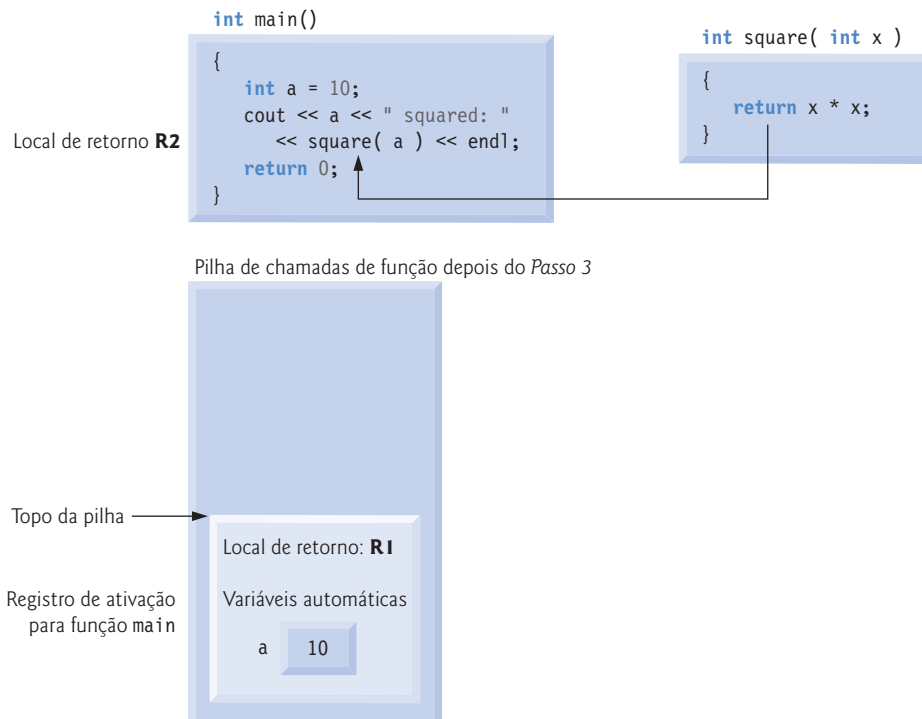
O significado de uma lista de parâmetros de função vazia em C++ é significativamente diferente daquele em C. Em C, significa que toda verificação de argumento está desativada (isto é, a chamada de função pode passar todos os argumentos que ela quiser). Em C++, significa que a função não aceita argumentos explicitamente. Portanto, os programas C que utilizam esse recurso podem causar erros de compilação quando compilados em C++.

Passo 2: main invoca square para realizar o cálculo.



**Figura 6.15** Pilha de chamadas de função depois de main invocar a função square para realizar o cálculo.

Passo 3: square retorna o resultado para main.



**Figura 6.16** Pilha de chamadas de função depois de a função square retornar para main.

```

1 // Figura 6.17: fig06_17.cpp
2 // Funções que não aceitam argumentos.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 void function1(); // função que não aceita argumentos
8 void function2(void); // função que não aceita argumentos
9
10 int main()
11 {
12 function1(); // chama function1 sem argumentos
13 function2(); // chama function2 sem argumentos
14 return 0; // indica terminação bem-sucedida
15 } // fim de main
16
17 // function1 utiliza uma lista de parâmetros vazia para especificar que
18 // a função não recebe argumentos
19 void function1()
20 {
21 cout << "function1 takes no arguments" << endl;
22 } // fim de function1
23
24 // function2 utiliza uma lista de parâmetros void para especificar que
25 // a função não recebe argumentos
26 void function2(void)
27 {
28 cout << "function2 also takes no arguments" << endl;
29 } // fim de function2

```

```

function1 takes no arguments
function2 also takes no arguments

```

**Figura 6.17** Funções que não aceitam argumentos.



### Erro comum de programação 6.13

*Os programas C++ não compilam a menos que protótipos de funções sejam oferecidos a cada função ou que cada função seja definida antes de ser chamada.*

## 6.13 Funções inline

Implementar um programa como um conjunto de funções é bom do ponto de vista da engenharia de software, mas as chamadas de função envolvem overhead de tempo de execução. O C++ fornece **funções inline** para ajudar a reduzir o overhead de chamada de função — especialmente para funções pequenas. Colocar o qualificador **inline** antes do tipo de retorno de uma função na definição de função ‘aconselha’ o compilador a gerar uma cópia do código da função no seu lugar (quando apropriado) para evitar uma chamada de função. Em troca, múltiplas cópias do código de função são inseridas no programa (tornando freqüentemente o programa maior) em vez de haver uma única cópia da função para a qual o controle é passado toda vez que a função é chamada. O compilador pode ignorar o qualificador **inline** e em geral faz isso para todas as funções, exceto as menores.



### Observação de engenharia de software 6.11

*Qualquer alteração em uma função **inline** poderia exigir que todos os clientes da função fossem recompilados. Isso pode ser significativo em algumas situações de desenvolvimento e manutenção de programas.*

**Boa prática de programação 6.5**

*O qualificador `inline` deve ser utilizado somente com funções pequenas, frequentemente utilizadas.*

**Dica de desempenho 6.4**

*Utilizar funções `inline` pode reduzir o tempo de execução, mas pode aumentar o tamanho do programa.*

A Figura 6.18 utiliza a função `inline` `cube` (linhas 11–14) para calcular o volume de um cubo de lado `side`. A palavra-chave `const` na lista de parâmetros da função `cube` (linha 11) informa ao compilador que a função não modifica a variável `side`. Isso assegura que o valor de `side` não é alterado pela função quando o cálculo é realizado. (A palavra-chave `const` é discutida em detalhes nos capítulos 7, 8 e 10.) Note que a definição completa da função `cube` aparece antes de ela ser utilizada no programa. Isso é necessário para que o compilador saiba expandir uma chamada de função `cube` dentro de seu código `inline`. Por essa razão, as funções `inline` reutilizáveis são normalmente colocadas em arquivos de cabeçalho, para que suas definições possam ser incluídas em cada arquivo-fonte que as utiliza.

**Observação de engenharia de software 6.12**

*O qualificador `const` deve ser utilizado para impor o princípio do menor privilégio. Utilizar o princípio do menor privilégio para projetar o software de modo adequado pode reduzir significativamente o tempo de depuração e efeitos colaterais, e pode tornar um programa mais fácil de modificar e manter.*

**6.14 Referências e parâmetros de referência**

Duas maneiras de passar argumentos para funções em muitas linguagens de programação são **passagem por valor** e **passagem por referência**. Quando um argumento é passado por valor, uma *cópia* do valor do argumento é feita e passada (na pilha de chamadas de função) para a função chamada. As alterações na cópia não afetam o valor da variável original no chamador. Isso previne os efeitos

```

1 // Figura 6.18: fig06_18.cpp
2 // Utilizando uma função inline para calcular o volume de um cubo.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 // Definição da função inline cube. A definição de função aparece antes
9 // de a função ser chamada, então um protótipo de função não é necessário.
10 // A primeira linha da definição de função atua como o protótipo.
11 inline double cube(const double side)
12 {
13 return side * side * side; // calcula o cubo
14 } // fim da função cube
15
16 int main()
17 {
18 double sideValue; // armazena o valor inserido pelo usuário
19 cout << "Enter the side length of your cube: ";
20 cin >> sideValue; // lê o valor fornecido pelo usuário
21
22 // calcula o cubo de sideValue e exibe o resultado
23 cout << "Volume of cube with side "
24 << sideValue << " is " << cube(sideValue) << endl;
25 return 0; // indica terminação bem-sucedida
26 } // fim de main

```

```

Enter the side length of your cube: 3.5
Volume of cube with side 3.5 is 42.875

```

**Figura 6.18** Função `inline` que calcula o volume de um cubo.

colaterais acidentais que tanto impedem o desenvolvimento de sistemas de software confiáveis e corretos. Até agora, cada argumento passado nos programas deste capítulo foi passado por valor.



### Dica de desempenho 6.5

*Uma desvantagem de passar por valor é que, se um item de dados grande estiver sendo passado, copiar esses dados pode exigir uma quantidade considerável de tempo de execução e espaço de memória.*

#### Parâmetros de referência

Esta seção introduz **parâmetros de referência** — a primeira de duas maneiras como o C++ permite passagem por referência. Com a passagem por referência, o chamador fornece à função chamada a capacidade de acessar os dados do chamador diretamente e modificar esses dados se a função chamada escolher fazer isso.



### Dica de desempenho 6.6

*A passagem por referência é boa por razões de desempenho, porque pode eliminar o overhead da passagem por valor de copiar grandes quantidades de dados.*



### Observação de engenharia de software 6.13

*A passagem por referência pode enfraquecer a segurança, porque a função chamada pode corromper os dados do chamador.*

Mais adiante, mostraremos como alcançar a vantagem de desempenho da passagem por referência enquanto alcançamos, simultaneamente, a vantagem de engenharia de software de proteger os dados do chamador contra corrupção.

Um parâmetro de referência é um *alias* (um ‘apelido’, pronuncia-se ‘álías’) para seu argumento correspondente em uma chamada de função. Para indicar que um parâmetro de função é passado por referência, simplesmente coloque um ‘e comercial’ (&) depois do tipo do parâmetro no protótipo de função; use a mesma convenção ao listar o tipo do parâmetro no cabeçalho de função. Por exemplo, a seguinte declaração em um cabeçalho de função

```
int &count
```

quando lida da direita para a esquerda é pronunciada ‘count é uma referência para um int’. Na chamada de função, simplesmente mencione a variável por nome para passá-la por referência. Então, mencionar a variável por seu nome de parâmetro no corpo da função chamada na realidade referencia a variável original na função chamadora; e a variável original pode ser modificada diretamente pela função chamada. Como sempre, o protótipo de função e o cabeçalho devem ser correspondentes.

#### Passando argumentos por valor e por referência

A Figura 6.19 compara a passagem por valor e por referência com os parâmetros de referência. Os ‘estilos’ dos argumentos nas chamadas para as funções `squareByValue` e `squareByReference` são idênticos — ambas as variáveis são simplesmente mencionadas por nome nas chamadas de função. Sem verificar os protótipos de função ou definições de função, não é possível informar, considerando as chamadas isoladamente, se alguma função pode modificar seus argumentos. Mas como os protótipos de função são obrigatórios, o compilador não tem problemas para resolver a ambigüidade.



### Erro comum de programação 6.14

*Como os parâmetros de referência são mencionados apenas pelo nome no corpo da função chamada, o programador poderia inadvertidamente tratar os parâmetros de referência como parâmetros passados por valor. Isso pode causar efeitos colaterais inesperados se as cópias originais das variáveis forem alteradas pela função.*

O Capítulo 8 discute ponteiros; estes permitem uma forma alternativa de passagem por referência na qual o estilo da chamada indica claramente a passagem por referência (e o potencial para modificar os argumentos do chamador).



### Dica de desempenho 6.7

*Para passar objetos grandes, utilize um parâmetro de referência constante a fim de simular a aparência e a segurança da passagem por valor e evitar o overhead de passar uma cópia do objeto grande.*



### Observação de engenharia de software 6.14

*Muitos programadores não se incomodam em declarar parâmetros passados por valor como `const`, mesmo que a função chamada não deva modificar o argumento passado. A palavra-chave `const` nesse contexto protegeria apenas uma cópia do argumento original, não o próprio argumento original, que, quando passado por valor, é protegido contra modificação pela função chamada.*

```

1 // Figura 6.19: fig06_19.cpp
2 // Comparando a passagem por valor e a passagem por referência com as referências.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int squareByValue(int); // protótipo de função (passagem por valor)
8 void squareByReference(int &); // protótipo de função (passagem por referência)
9
10 int main()
11 {
12 int x = 2; // valor para square utilizando squareByValue
13 int z = 4; // valor para square utilizando squareByReference
14
15 // demonstra squareByValue
16 cout << "x = " << x << " before squareByValue\n";
17 cout << "Value returned by squareByValue: "
18 << squareByValue(x) << endl;
19 cout << "x = " << x << " after squareByValue\n" << endl;
20
21 // demonstra squareByReference
22 cout << "z = " << z << " before squareByReference" << endl;
23 squareByReference(z);
24 cout << "z = " << z << " after squareByReference" << endl;
25 return 0; // indica terminação bem-sucedida
26 } // fim de main
27
28 // squareByValue multiplica um número por ele próprio, armazena o
29 // resultado em number e retorna o novo valor de number
30 int squareByValue(int number)
31 {
32 return number *= number; // argumento do chamador não modificado
33 } // fim da função squareByValue
34
35 // squareByReference multiplica numberRef por si mesmo e armazena o resultado
36 // na variável à qual numberRef se refere na função main
37 void squareByReference(int &numberRef)
38 {
39 numberRef *= numberRef; // argumento do chamador modificou
40 } // fim da função squareByReference

```

```

x = 2 before squareByValue
Value returned by squareByValue: 4
x = 2 after squareByValue

z = 4 before squareByReference
z = 16 after squareByReference

```

**Figura 6.19** Passando argumentos por valor e por referência.

Para especificar uma referência a uma constante, coloque o qualificador `const` antes do especificador de tipo na declaração de parâmetro.

Observe na linha 37 da Figura 6.19 a colocação de `&` na lista de parâmetros de função `squareByReference`. Alguns programadores em C++ preferem escrever `int& numberRef`.





### Observação de engenharia de software 6.15

*Pelas razões combinadas de clareza e desempenho, muitos programadores em C++ preferem que argumentos modificáveis sejam passados para as funções utilizando ponteiros (que estudamos no Capítulo 8), pequenos argumentos não modificáveis sejam passados por valor e argumentos grandes não modificáveis sejam passados para funções utilizando referências a constantes.*

#### Referências como aliases dentro de uma função

As referências também podem ser utilizadas como aliases para outras variáveis dentro de uma função (embora, em geral, sejam utilizadas com funções, como mostrado na Figura 6.19). Por exemplo, o código

```
int count = 1; // declara a variável count do tipo inteiro
int &cRef = count; // cria cRef como um alias para count
cRef++; // incrementa count (utilizando seu alias cRef)
```

incrementa a variável count utilizando seu alias cRef. As variáveis de referência devem ser inicializadas em suas declarações (ver figuras 6.20 e 6.21) e não podem ser reatribuídas como aliases a outras variáveis. Uma vez que uma referência é declarada como um alias para outra variável, todas as operações supostamente realizadas no alias (isto é, a referência) são realmente realizadas na variável original. O alias tem simplesmente outro nome para a variável original. Aceitar o endereço de uma referência e comparar referências não produzem erros de sintaxe; em vez disso, cada operação ocorre na variável para a qual a referência é um alias. A menos que seja referência a uma constante, um argumento de referência deve ser um *lvalue* (por exemplo, um nome variável), não uma constante ou expressão que retorna um *rvalue* (por exemplo, o resultado de um cálculo). Veja a Seção 5.9 para obter definições dos termos *lvalue* e *rvalue*.

#### Retornando uma referência de uma função

As funções podem retornar referências, mas isso pode ser perigoso. Ao retornar referência a uma variável declarada na função chamada, a variável deve ser declarada `static` dentro dessa função. Caso contrário, a referência referencia uma variável automática que é descartada quando a função termina; diz-se que essa variável é 'indefinida' e o comportamento do programa é imprevisível. As referências a variáveis indefinidas são chamadas **referências oscilantes**.



### Erro comum de programação 6.15

*Não inicializar uma variável de referência quando ela é declarada é um erro de compilação, a menos que a declaração faça parte da lista de parâmetros de uma função. Os parâmetros de referência são inicializados quando a função em que são declarados é chamada.*

```
1 // Figura 6.20: fig06_20.cpp
2 // As referências devem ser inicializadas.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9 int x = 3;
10 int &y = x; // y referencia (é um alias para) x
11
12 cout << "x = " << x << endl << "y = " << y << endl;
13 y = 7; // realmente modifica x
14 cout << "x = " << x << endl << "y = " << y << endl;
15 return 0; // indica terminação bem-sucedida
16 } // fim de main
```

```
x = 3
y = 3
x = 7
y = 7
```

**Figura 6.20** Inicializando e utilizando uma referência.

```

1 // Figura 6.21: fig06_21.cpp
2 // As referências devem ser inicializadas.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9 int x = 3;
10 int &y; // Erro: y deve ser inicializado
11
12 cout << "x = " << x << endl << "y = " << y << endl;
13 y = 7;
14 cout << "x = " << x << endl << "y = " << y << endl;
15 return 0; // indica terminação bem-sucedida
16 } // fim de main

```

Mensagem de erro do compilador de linha de comando da Borland C++:

```

Error E2304 C:\cphttp5_examples\ch06\Fig06_21\fig06_21.cpp 10:
Reference variable 'y' must be initialized in function main()

```

Mensagem de erro do compilador Microsoft Visual C++:

```

C:\cphttp5_examples\ch06\Fig06_21\fig06_21.cpp(10) : error C2530: 'y' :
references must be initialized

```

Mensagem de erro do compilador GNU C++:

```

fig06_21.cpp:10: error: 'y' declared as a reference but not initialized

```

**Figura 6.21** Referência não inicializada produz um erro de sintaxe.



## Erro comum de programação 6.16

Tentar reatribuir uma referência anteriormente declarada como um alias de outra variável é um erro de lógica. O valor da outra variável é simplesmente atribuído à variável para a qual a referência já é um alias.



## Erro comum de programação 6.17

Retornar uma referência a uma variável automática em uma função chamada é um erro de lógica. Alguns compiladores emitem um aviso quando isso ocorre.

### Mensagens de erro para referências não inicializadas

Observe que o padrão C++ não especifica as mensagens de erro que os compiladores utilizam para indicar erros particulares. Por essa razão, a Figura 6.21 mostra as mensagens de erro produzidas pelo compilador de linha de comando Borland C++ 5.5, pelo compilador Microsoft Visual C++ .NET e pelo compilador GNU C++ quando uma referência não é inicializada.

## 6.15 Argumentos-padrão

Não é incomum que um programa invoque uma função repetidamente com o mesmo valor de argumento para um parâmetro particular. Nesses casos, o programador pode especificar que tal parâmetro tem um **argumento-padrão**, isto é, um valor-padrão a ser passado a esse parâmetro. Quando um programa omite um argumento para um parâmetro com um argumento-padrão em uma chamada de função, o compilador reescreve a chamada de função e insere o valor-padrão desse argumento a ser passado como um argumento para a chamada de função.

Argumentos-padrão devem ser os argumentos mais à direita (finais) em uma lista de parâmetros da função. Quando chamada uma função com dois argumentos-padrão, se o argumento omitido não for o argumento mais à direita na lista de argumentos, então todos

os argumentos à direita desse argumento devem também ser omitidos. Os argumentos-padrão devem ser especificados com a primeira ocorrência do nome de função — em geral, no protótipo de função. Se o protótipo de função é omitido porque a definição de função também serve como o protótipo, então os argumentos-padrão devem ser especificados no cabeçalho de função. Os valores-padrão podem ser qualquer expressão, inclusive constantes, variáveis globais ou chamadas de função. Os argumentos-padrão também podem ser utilizados com funções inline.

A Figura 6.22 demonstra como utilizar argumentos-padrão no cálculo do volume de uma caixa. O protótipo de função para `boxVolume` (linha 8) especifica que todos os três parâmetros receberam valores-padrão de 1. Observe que fornecemos nomes de variável no protótipo de função por uma questão de legibilidade. Como sempre, os nomes de variável não são necessários nos protótipos de função.

```

1 // Figura 6.22: fig06_22.cpp
2 // Utilizando argumentos-padrão.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 // protótipo de função que especifica argumentos-padrão
8 int boxVolume(int length = 1, int width = 1, int height = 1);
9
10 int main()
11 {
12 // nenhum argumento – utilize valores-padrão para todas as dimensões
13 cout << "The default box volume is: " << boxVolume();
14
15 // especifica o comprimento; largura e altura-padrão
16 cout << "\n\nThe volume of a box with length 10,\n"
17 << "width 1 and height 1 is: " << boxVolume(10);
18
19 // especifica comprimento e largura; altura-padrão
20 cout << "\n\nThe volume of a box with length 10,\n"
21 << "width 5 and height 1 is: " << boxVolume(10, 5);
22
23 // especifica todos os argumentos
24 cout << "\n\nThe volume of a box with length 10,\n"
25 << "width 5 and height 2 is: " << boxVolume(10, 5, 2);
26 << endl;
27 return 0; // indica terminação bem-sucedida
28 } // fim de main
29
30 // função boxVolume calcula o volume de uma caixa
31 int boxVolume(int length, int width, int height)
32 {
33 return length * width * height;
34 } // fim da função boxVolume

```

The default box volume is: 1

The volume of a box with length 10,  
width 1 and height 1 is: 10

The volume of a box with length 10,  
width 5 and height 1 is: 50

The volume of a box with length 10,  
width 5 and height 2 is: 100

**Figura 6.22** Argumentos-padrão para uma função.

**Erro comum de programação 6.18**

*É um erro de compilação especificar argumentos-padrão no protótipo e no cabeçalho da função.*

A primeira chamada a `boxVolume` (linha 13) não especifica nenhum argumento, utilizando assim os três valores-padrão de 1. A segunda chamada (linha 17) passa um argumento `length`, utilizando assim valores-padrão de 1 para os argumentos `width` e `height`. A terceira chamada (linha 21) passa os argumentos `width` e `height`, utilizando assim um valor-padrão de 1 para o argumento `height`. A última chamada (linha 25) passa argumentos para `length`, `width` e `height`, utilizando assim valores não-padrão. Observe que quaisquer argumentos passados à função explicitamente são atribuídos aos parâmetros da função da esquerda para a direita. Portanto, quando `boxVolume` recebe um argumento, a função atribui o valor desse argumento ao seu parâmetro `length` (isto é, o parâmetro mais à esquerda na lista de parâmetros). Quando `boxVolume` recebe dois argumentos, a função atribui os valores desses argumentos a seus parâmetros `length` e `width` nessa ordem. Por fim, quando `boxVolume` receber os três argumentos, a função atribui o valor desse argumento a seus parâmetros `length`, `width` and `height`, respectivamente.

**Boa prática de programação 6.6**

*Utilizar argumentos-padrão pode simplificar a escrita de chamadas de função. Entretanto, alguns programadores sentem que é mais claro especificar todos os argumentos explicitamente.*

**Observação de engenharia de software 6.16**

*Se os valores-padrão de uma função mudam, todo o código-cliente utilizando a função deve ser recompilado.*

**Erro comum de programação 6.19**

*Especificar e tentar utilizar um argumento-padrão que não é um argumento mais à direita (final) (enquanto não simultaneamente assumindo o padrão para todos os argumentos mais à direita) é um erro de sintaxe.*

**6.16 Operador de solução de escopo unário**

É possível declarar variáveis locais e globais do mesmo nome. O C++ fornece o **operador unário de resolução de escopo (::)** para acessar uma variável global quando uma variável local do mesmo nome estiver no escopo. O operador unário de resolução de escopo não pode ser utilizado para acessar uma variável local do mesmo nome em um bloco externo. Uma variável global pode ser acessada diretamente sem o operador unário de resolução de escopo se o nome da variável global não for o mesmo de uma variável local no escopo.

A Figura 6.23 demonstra o operador unário de resolução de escopo com variáveis locais e globais do mesmo nome (linhas 7 e 11). Para enfatizar que versões locais e globais de número variável são distintas, o programa declara uma variável de tipo `int` e a outra `double`.

Utilizar o operador unário de resolução de escopo (::) com um nome variável dado é opcional quando a única variável com esse nome é uma variável global.

**Erro comum de programação 6.20**

*É um erro tentar utilizar o operador unário de resolução de escopo (::) para acessar uma variável não global em um bloco externo. Se não existir nenhuma variável global com esse nome, ocorre um erro de compilação. Se existir uma variável global com esse nome, esse é um erro de lógica, porque o programa referenciará a variável global quando pretendia acessar a variável não global no bloco externo.*

**Boa prática de programação 6.7**

*Sempre utilizar o operador unário de resolução de escopo (::) para referenciar as variáveis globais torna os programas mais fáceis de ler e entender, porque torna claro que você está pretendendo acessar uma variável global em vez de uma variável não global.*

**Observação de engenharia de software 6.17**

*Utilizar sempre o operador unário de resolução de escopo (::) para referenciar a variáveis globais torna os programas mais fáceis de modificar, reduzindo o risco de colisões de nome com as variáveis não globais.*

**Dica de prevenção de erro 6.4**

*Sempre utilizar o operador unário de resolução de escopo (::) para referenciar uma variável global elimina possíveis erros de lógica que podem ocorrer se uma variável não global ocultar a variável global.*

```

1 // Figura 6.23: fig06_23.cpp
2 // Utilizando o operador unário de resolução de escopo.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int number = 7; // variável global chamada number
8
9 int main()
10 {
11 double number = 10.5; // variável local chamada number
12
13 // exibe valores de variáveis locais e globais
14 cout << "Local double value of number = " << number
15 << "\nGlobal int value of number = " << ::number << endl;
16 return 0; // indica terminação bem-sucedida
17 } // fim de main

```

```

Local double value of number = 10.5
Global int value of number = 7

```

**Figura 6.23** Operador unário de resolução de escopo.



### Dica de prevenção de erro 6.5

Evite utilizar variáveis do mesmo nome para propósitos diferentes em um programa. Embora isso seja permitido em várias circunstâncias, pode levar a erros.

## 6.17 Sobrecarga de funções

O C++ permite que várias funções do mesmo nome sejam definidas, contanto que essas funções tenham conjuntos diferentes de parâmetros (pelo menos no que diz respeito aos tipos de parâmetro ou o número de parâmetros, ou à ordem dos tipos de parâmetro). Essa capacidade é chamada de **sobrecarga de funções**. Quando uma função sobrecarregada é chamada, o compilador C++ seleciona a função adequada examinando o número, os tipos e a ordem dos argumentos na chamada. A sobrecarga de funções é comumente utilizada para criar várias funções do mesmo nome que realizam tarefas semelhantes, mas em tipos de dados diferentes. Por exemplo, muitas funções na biblioteca de matemática são sobrecarregadas para tipos de dados diferentes.<sup>1</sup>



### Boa prática de programação 6.8

Sobrecarregar funções que realizam tarefas intimamente relacionadas pode tornar os programas mais legíveis e compreensíveis.

#### Funções **square** sobrecarregadas

A Figura 6.24 utiliza funções **square** sobrecarregadas para calcular o quadrado de um `int` (linhas 8–12) e o quadrado de um `double` (linhas 15–19). A linha 23 invoca a versão `int` da função **square** passando o valor literal 7. O C++ trata o valor de número literal inteiro como tipo `int` por padrão. De maneira semelhante, a linha 25 invoca a versão `double` da função **square** passando o valor literal 7.5, que o C++ trata como um valor `double` por padrão. Em cada caso o compilador escolhe a chamada de função adequada, com base no tipo do argumento. As duas últimas linhas da janela de saída confirmam que a função adequada foi chamada em cada caso.

#### Como o compilador diferencia as funções sobrecarregadas

As funções sobrecarregadas são distinguidas por suas assinaturas. Uma assinatura é uma combinação de um nome da função e seus tipos de parâmetro (em ordem). O compilador codifica cada identificador de função com o número e os tipos de seus parâmetros (às vezes referidos como **desfiguração de nome** ou **decoração de nome**) para permitir **linkagem segura para tipos** (*type-safe linkage*). A linkagem segura para o tipo garante que a função sobrecarregada adequada seja chamada e que os tipos dos argumentos correspondam aos tipos dos parâmetros.

<sup>1</sup> O padrão C++ exige as versões sobrecarregadas das funções matemáticas `float`, `double` e `long double` da biblioteca de matemática discutidas na Seção 6.3.

```

1 // Figura 6.24: fig06_24.cpp
2 // Funções sobrecarregadas.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 // função square para valores int
8 int square(int x)
9 {
10 cout << "square of integer " << x << " is ";
11 return x * x;
12 } // fim da função square com argumento int
13
14 // função square para valores double
15 double square(double y)
16 {
17 cout << "square of double " << y << " is ";
18 return y * y;
19 } // fim da função square com argumento double
20
21 int main()
22 {
23 cout << square(7); // chama versão int
24 cout << endl;
25 cout << square(7.5); // chama versão double
26 cout << endl;
27 return 0; // indica terminação bem-sucedida
28 } // fim de main

```

```

square of integer 7 is 49
square of double 7.5 is 56.25

```

**Figura 6.24** Funções square sobrecarregadas.

A Figura 6.25 foi compilada com o compilador de linha de comando Borland C++ 5.6.4. Em vez de mostrar a saída de execução do programa (como normalmente faríamos), mostramos os nomes de função desfigurados produzidos na linguagem assembly do Borland C++. Cada nome desfigurado inicia com @ seguido pelo nome de função. O nome de função então é separado da lista de parâmetros desfigurados por \$q. Na lista de parâmetros da função `nothing2` (linha 25; veja a quarta linha de saída), `c` representa um `char`, `i` representa um `int`, `rf` representa um `float` & (isto é, uma referência a um `float`) e `rd` representa um `double` & (isto é, uma referência a um `double`). Na lista de parâmetros da função `nothing1`, `i` representa um `int`, `f` representa um `float`, `c` representa um `char` e `ri` representa um `int` &. As duas funções `square` são distinguidas por suas listas de parâmetros; uma especifica `d` para `double` e a outra especifica `i` para `int`. Os tipos de retorno das funções não são especificados nos nomes desfigurados. As funções sobrecarregadas podem ter tipos de retorno diferentes, mas se realmente tiverem, também devem ter listas de parâmetros diferentes. Novamente, você não pode ter duas funções com a mesma assinatura e diferentes tipos de retorno. Observe que a desfiguração de nome de função é específica do compilador. Observe também que a função `main` não é desfigurada, porque não pode ser sobrecarregada.



### Erro comum de programação 6.21

*Criar funções sobrecarregadas com listas de parâmetros idênticos e tipos de retorno diferentes é um erro de compilação.*

O compilador utiliza apenas as listas de parâmetros para distinguir entre funções do mesmo nome. As funções sobrecarregadas não precisam ter o mesmo número de parâmetros. Os programadores devem ter atenção ao sobrecarregar funções com parâmetros-padrão, porque isso pode causar ambigüidade.

#### Operadores sobrecarregados

No Capítulo 11, discutimos como sobrecarregar operadores para definir a maneira como eles devem operar em objetos de tipos de dados definidos pelo usuário. (De fato, utilizamos muitos operadores sobrecarregados até agora, incluindo o operador de inserção de fluxo `<<` e o de extração de fluxo `>>`, cada um dos quais é sobrecarregado para ser capaz de exibir dados de todos os tipos fundamentais. Falaremos

```

1 // Figura 6.25: fig06_25.cpp
2 // Desfiguração de nomes.
3
4 // função square para valores int
5 int square(int x)
6 {
7 return x * x;
8 } // fim da função square
9
10 // função square para valores double
11 double square(double y)
12 {
13 return y * y;
14 } // fim da função square
15
16 // função que recebe argumentos dos tipos
17 // int, float, char e int &
18 void nothing1(int a, float b, char c, int &d)
19 {
20 // esvazia o corpo da função
21 } // fim da função nothing1
22
23 // função que recebe argumentos dos tipos
24 // char, int, float & e double &
25 int nothing2(char a, int b, float &c, double &d)
26 {
27 return 0;
28 } // fim da função nothing2
29
30 int main()
31 {
32 return 0; // indica terminação bem-sucedida
33 } // fim de main

```

```

@square$qi
@square$qd
@nothing1$qifcri
@nothing2$qcirfrd
_main

```

**Figura 6.25** Desfiguração de nome para permitir linkagem segura para tipos.

mais sobre como sobrecarregar << e >> para ser capaz de tratar objetos de tipos definidos pelo usuário no Capítulo 11.) A Seção 6.18 introduz templates de função para gerar automaticamente as funções sobrecarregadas que realizam tarefas idênticas em tipos de dados diferentes.



## Erro comum de programação 6.22

*Uma função com argumentos-padrão omitidos poderia ser chamada de modo idêntico a outra função sobrecarregada; isso é um erro de compilação. Por exemplo, ter em um programa uma função que não aceita explicitamente nenhum argumento e uma função do mesmo nome que contém todos os argumentos-padrão resulta em um erro de compilação quando se tenta utilizar esse nome de função em uma chamada sem passar argumentos. O compilador não sabe que versão da função escolher.*

## 6.18 Templates de funções

Funções sobrecarregadas são normalmente utilizadas para realizar operações semelhantes que envolvem lógica de programa diferente em diferentes tipos de dados. Se a lógica e as operações do programa forem idênticas para cada tipo de dados, a sobrecarga pode ser realizada



```

1 // Figura 6.26: maximum.h
2 // Definição do template de função maximum.
3
4 template < class T > // ou template< typename T >
5 T maximum(T value1, T value2, T value3)
6 {
7 T maximumValue = value1; // pressupõe que value1 é máximo
8
9 // determina se value2 é maior que maximumValue
10 if (value2 > maximumValue)
11 maximumValue = value2;
12
13 // determina se value3 é maior que maximumValue
14 if (value3 > maximumValue)
15 maximumValue = value3;
16
17 return maximumValue;
18 } // fim do template de função maximum

```

**Figura 6.26** Arquivo de cabeçalho do template de função maximum.

de forma mais compacta e conveniente utilizando **templates de função**. O programador escreve uma única definição de template de função. Considerando os tipos de argumento fornecidos em chamadas para essa função, o C++ gera automaticamente **especializações de template de função** separadas para tratar cada tipo de chamada de maneira adequada. Portanto, definir um único template de função define essencialmente uma família inteira de funções sobrecarregadas.

A Figura 6.26 contém a definição de um template de função (linhas 4–18) para uma função maximum que determina o maior de três valores. Todas as definições de template de função iniciam com a palavra-chave `template` (linha 4) seguida por uma **lista de parâmetros de template** para o template de função entre colchetes angulares (< e >). Cada parâmetro na lista de parâmetros do template (frequentemente referido como um **parâmetro de tipo formal**) é precedido pela palavra-chave `typename` ou pela palavra-chave `class` (que são sinônimas). Os parâmetros de tipo formais são marcadores de lugar para tipos fundamentais ou tipos definidos pelo usuário. Esses marcadores de lugar são utilizados para especificar os tipos dos parâmetros da função (linha 5), especificar o tipo de retorno da função (linha 5) e declarar variáveis dentro do corpo da definição de função (linha 7). Um template de função é definido como qualquer outra função, mas utiliza os parâmetros formais de tipo como marcadores de lugar para tipos de dados reais.

O template de função na Figura 6.26 declara um único parâmetro formal do tipo `T` (linha 4) como um marcador de lugar para o tipo dos dados a ser testado pela função maximum. O nome de um parâmetro de tipo deve ser único na lista de parâmetros de template para uma definição de template particular. Quando o compilador detecta uma invocação de `maximum` no código-fonte do programa, o tipo dos dados passado para `maximum` é substituído por `T` em toda a definição de template, e o C++ cria uma função completa para determinar o máximo de três valores do tipo de dados especificado. Então a função recém-criada é compilada. Portanto, os templates são um meio de gerar código.



### Erro comum de programação 6.23

*Não colocar a palavra-chave `class` ou `typename` antes de cada parâmetro de tipo formal de um template de função (por exemplo, escrever `< class S, T >` em vez de `< class S, class T >`) é um erro de sintaxe.*

A Figura 6.27 utiliza o template de função maximum (linhas 20, 30 e 40) para determinar o maior de três valores `int`, três valores `double` e três valores `char`.

```

1 // Figura 6.27: fig06_27.cpp
2 // Programa de teste do template de função maximum.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;

```

**Figura 6.27** Demonstrando o template de função maximum.

(continua)

```

7
8 #include "maximum.h" // inclui a definição do template de função maximum
9
10 int main()
11 {
12 // demonstra maximum com valores int
13 int int1, int2, int3;
14
15 cout << "Input three integer values: ";
16 cin >> int1 >> int2 >> int3;
17
18 // invoca a versão int de maximum
19 cout << "The maximum integer value is: "
20 << maximum(int1, int2, int3);
21
22 // demonstra maximum com valores double
23 double double1, double2, double3;
24
25 cout << "\n\nInput three double values: ";
26 cin >> double1 >> double2 >> double3;
27
28 // invoca a versão double de maximum
29 cout << "The maximum double value is: "
30 << maximum(double1, double2, double3);
31
32 // demonstra maximum com valores char
33 char char1, char2, char3;
34
35 cout << "\n\nInput three characters: ";
36 cin >> char1 >> char2 >> char3;
37
38 // invoca versão char de maximum
39 cout << "The maximum character value is: "
40 << maximum(char1, char2, char3) << endl;
41 return 0; // indica terminação bem-sucedida
42 } // fim de main

```

Input three integer values: 1 2 3

The maximum integer value is: 3

Input three double values: 3.3 2.2 1.1

The maximum double value is: 3.3

Input three characters: A C B

The maximum character value is: C

**Figura 6.27** Demonstrando o template de função maximum.

(continuação)

Na Figura 6.27, três funções são criadas como resultado das chamadas nas linhas 20, 30 e 40 — esperando três valores `int`, três valores `double` e três valores `char`, respectivamente. A especialização de template de função criada para o tipo `int` substitui cada ocorrência de `T` por `int` como mostrado a seguir:

```

int maximum(int value1, int value2, int value3)
{
 int maximumValue = value1;

 // determina se value2 é maior que maximumValue

```

```

if (value2 > maximumValue)
 maximumValue = value2;

// determina se value3 é maior que maximumValue
if (value3 > maximumValue)
 maximumValue = value3;

return maximumValue;
} // fim do template de função maximum

```

## 6.19 Recursão

Os programas que discutimos geralmente são estruturados como funções que chamam umas às outras de uma maneira hierárquica, disciplinada. Para alguns problemas, é útil ter as funções chamando umas às outras. Uma **função recursiva** é uma função que chama a si mesma, direta ou indiretamente (por outra função).<sup>2</sup> A recursão é um tópico importante, discutido detalhadamente em cursos de nível superior de ciência da computação. Esta seção e a próxima apresentam exemplos simples de recursão. Este livro contém um extenso tratamento da recursão. A Figura 6.33 (no final da Seção 6.21) resume os exemplos e exercícios de recursão no livro.

Primeiro consideramos a recursão conceitualmente e, em seguida, examinamos dois programas contendo funções recursivas. Abordagens de solução de problemas de recursão têm um número de elementos em comum. Uma função recursiva é chamada para resolver um problema. A função realmente sabe como resolver somente o(s) caso(s) mais simples, ou os chamado(s) **caso(s) básico(s)**. Se a função é chamada com um caso básico, ela simplesmente retorna um resultado. Se a função é chamada com um problema mais complexo, em geral, ela divide o problema em duas partes conceituais — uma parte que a função sabe fazer e outra que não sabe. Para tornar a recursão realizável, a última parte deve parecer-se com o problema original, mas ser uma versão ligeiramente mais simples ou ligeiramente menor. Esse novo problema é parecido com o problema original; portanto, a função carrega (chama) uma cópia nova dela própria para trabalhar no problema menor — isso é referido como **chamada recursiva** e também é chamado de **passo de recursão**. O passo de recursão freqüentemente inclui a palavra-chave `return`, porque seu resultado será combinado com a parte do problema que a função soube resolver para formar um resultado que será passado de volta ao chamador original, possivelmente `main`.

O passo de recursão executa enquanto a chamada original à função ainda está aberta, isto é, não terminou de executar. O passo de recursão pode resultar em muito mais dessas chamadas recursivas, uma vez que a função continua dividindo cada novo subproblema com o qual a função é chamada em duas partes conceituais. Para que a recursão, por fim, termine, toda vez que a função chamar a si mesma com uma versão ligeiramente mais simples do problema original, essa seqüência de problemas cada vez menor deve, finalmente, convergir para o caso básico. Nesse ponto, a função reconhece o caso básico e retorna um resultado à cópia anterior da função, e uma seqüência de retornos se segue até que a chamada de função original por fim retorne o resultado final para `main`. Tudo isso soa bem estranho comparado ao tipo de resolução de problemas ‘convencional’ que utilizamos até agora. Como um exemplo desses conceitos em operação, vamos escrever um programa recursivo para realizar um cálculo matemático popular.

O fatorial de um inteiro  $n$  não negativo, escrito  $n!$  (e pronunciado como ‘ $n$  fatorial’), é o produto

$$n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 1$$

com  $1!$  igual a 1, e  $0!$  definido como 1. Por exemplo,  $5!$  é o produto  $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$ , que é igual a 120.

O fatorial de um inteiro, `number`, maior que ou igual a 0, pode ser calculado **iterativamente** (não recursivamente) utilizando uma instrução `for` como mostrado a seguir:

```

factorial = 1;

for (int counter = number; counter >= 1; counter--)
 factorial *= counter;

```

Chega-se a uma definição recursiva da função fatorial observando o seguinte relacionamento:

$$n! = n \cdot (n-1)!$$

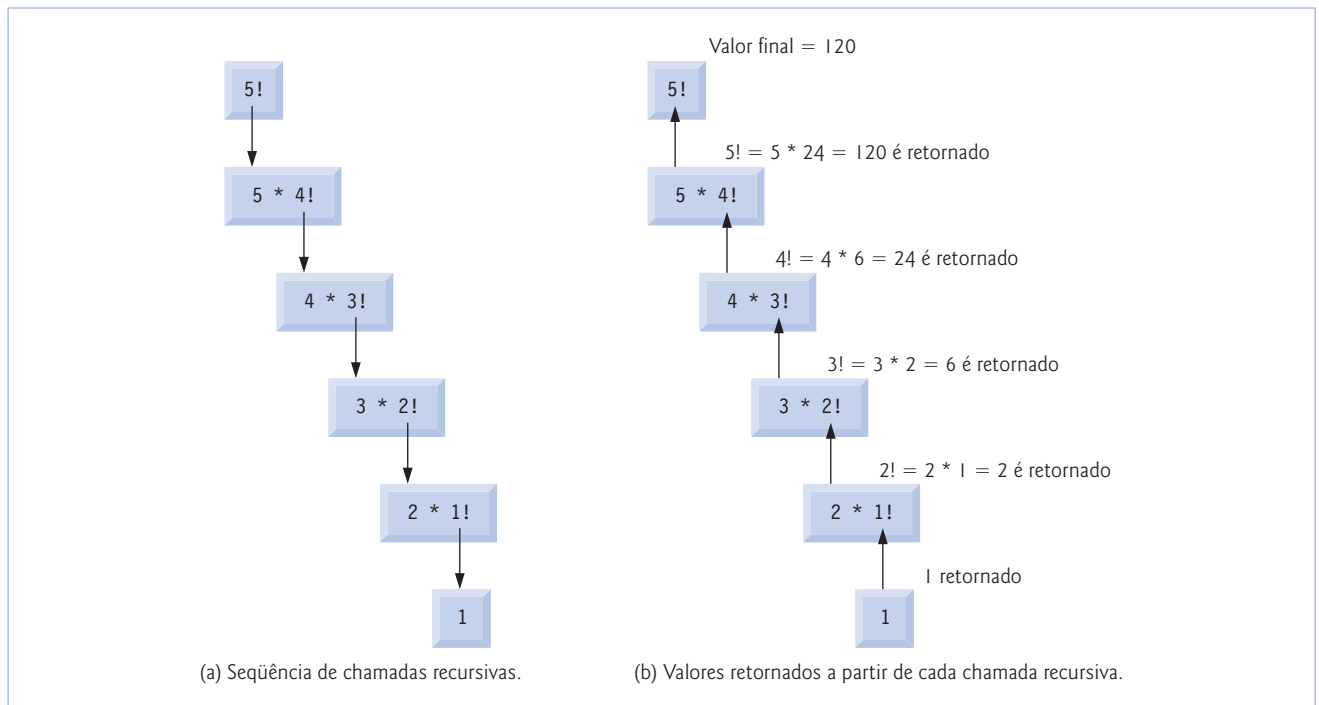
Por exemplo,  $5!$  é claramente igual a  $5 \cdot 4!$ . Como mostrado a seguir:

$$\begin{aligned}
 5! &= 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 \\
 5! &= 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1) \\
 5! &= 5 \cdot (4!)
 \end{aligned}$$

A avaliação de  $5!$  prosseguiria como mostrado na Figura 6.28. A Figura 6.28(a) mostra como a sucessão de chamadas recursivas prossegue até  $1!$ , que é avaliado como 1, o que termina a recursão. A Figura 6.28(b) mostra os valores retornados de cada chamada recursiva para seu chamador até que o valor final seja calculado e retornado.

O programa da Figura 6.29 utiliza recursão para calcular e imprimir o fatorial dos inteiros 0–10. (A escolha do tipo de dados `unsigned long` é explicada daqui a pouco.) A função recursiva `factorial` (linhas 23–29) primeiro determina se a condição de terminação

<sup>2</sup> Embora muitos compiladores permitam que a função `main` chame a si própria, a Seção 3.6.1, parágrafo 3, da documentação do padrão C++ indica que `main` não deve ser chamado a partir de dentro de um programa. Seu único propósito é ser o ponto inicial para a execução de programa.



**Figura 6.28** Avaliação recursiva de  $5!$ .

`number <= 1` (linha 25) é verdadeira. Se `number` for de fato menor que ou igual a 1, a função `factorial` retornará 1 (linha 26), nenhuma recursão adicional será necessária e a função terminará. Se `number` for maior que 1, a linha 28 expressa o problema como o produto de `number` e uma chamada recursiva para `factorial` avaliar o fatorial de `number - 1`. Observe que `factorial( number - 1 )` é um problema ligeiramente mais simples que o cálculo original `factorial( number )`.

A função `factorial` foi declarada para receber um parâmetro do tipo `unsigned long` e retornar um resultado de tipo `unsigned long`. Essa é a notação abreviada para `unsigned long int`. A documentação do padrão C++ requer que uma variável de tipo `unsigned long int` seja armazenada em pelo menos quatro bytes (32 bits); portanto, ela pode armazenar um valor no intervalo de 0 a pelo menos 4.294.967.295. (O tipo de dados `long int` também é armazenado em pelo menos quatro bytes e pode armazenar um valor pelo menos no intervalo de -2.147.483.648 a 2.147.483.647.) Como pode ser visto na Figura 6.29, valores fatoriais tornam-se grandes rapidamente. Escolhemos o tipo de dados `unsigned long` para que o programa possa calcular fatoriais maiores que  $7!$  em computadores com inteiros pequenos (como dois bytes). Infelizmente, a função `factorial` produz valores grandes com tanta rapidez que até `unsigned long` não nos ajuda a calcular muitos valores fatoriais antes mesmo de o tamanho de uma variável `unsigned long` ser excedido.

Os exercícios exploram o uso de variáveis do tipo de dados `double` para calcular fatoriais de números maiores. Isso aponta para uma fraqueza na maioria das linguagens de programação, a saber, que as linguagens não são facilmente estendidas para tratar os requisitos únicos de vários aplicativos. Como veremos ao discutir a programação orientada a objetos em maior profundidade, o C++ é uma linguagem extensível que permite criar classes que podem representar inteiros arbitrariamente grandes se quisermos. Tais classes já estão disponíveis em bibliotecas de classes populares,<sup>3</sup> e trabalhamos em classes semelhantes nos exercícios 9.14 e 11.5.



### Erro comum de programação 6.24

*Omitir o caso básico ou escrever o passo de recursão incorretamente de modo que ele não convirja para o caso básico causa recursão 'infinita', esgotando por fim a memória. Isso é análogo ao problema de um loop infinito em uma solução iterativa (não recursiva).*

## 6.20 Exemplo que utiliza recursão: série de Fibonacci

A série de Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

inicia com 0 e 1 e tem a propriedade de que cada número de Fibonacci subsequente é a soma dos dois números de Fibonacci anteriores.

<sup>3</sup> Essas classes podem ser encontradas em [shoup.net/ntl](http://shoup.net/ntl), [cliodhna.cop.uop.edu/~hetrick/c-sources.html](http://cliodhna.cop.uop.edu/~hetrick/c-sources.html) e [www.trumphurst.com/cpplib/databpage.phtml?category='intro'](http://www.trumphurst.com/cpplib/databpage.phtml?category='intro').

```

1 // Figura 6.29: fig06_29.cpp
2 // Testando a função fatorial recursiva.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 unsigned long factorial(unsigned long); // protótipo de função
11
12 int main()
13 {
14 // calcula o fatorial de 0 a 10
15 for (int counter = 0; counter <= 10; counter++)
16 cout << setw(2) << counter << "! = " << factorial(counter)
17 << endl;
18
19 return 0; // indica terminação bem-sucedida
20 } // fim de main
21
22 // definição recursiva da função fatorial
23 unsigned long factorial(unsigned long number)
24 {
25 if (number <= 1) // testa caso básico
26 return 1; // casos básicos: 0! = 1 e 1! = 1
27 else // passo de recursão
28 return number * factorial(number - 1);
29 } // fim da função fatorial

```

```

0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800

```

**Figura 6.29** Demonstrando a função factorial.

A série ocorre na natureza e, em particular, descreve a forma de uma espiral. A relação de números de Fibonacci sucessivos converge para um valor constante de 1,618.... Esse número, também, ocorre freqüentemente na natureza e é chamado de **relação áurea** ou **média áurea**. Humanos tendem a achar a média áurea esteticamente agradável. Os arquitetos freqüentemente projetam janelas, salas e edifícios cujo comprimento e largura estão na relação da média áurea. Os cartões-postais freqüentemente são projetados com uma relação de comprimento/largura da média áurea.

A série de Fibonacci pode ser definida recursivamente como segue:

```

fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(n) = fibonacci(n - 1) + fibonacci(n - 2)

```

O programa da Figura 6.30 calcula o  $n$ -ésimo número de Fibonacci recursivamente utilizando a função `fibonacci`. Note que os números de Fibonacci também tendem a se tornar rapidamente grandes, embora mais lentamente do que os fatoriais. Portanto, escolhamos o tipo de dados `unsigned long` para o tipo de parâmetro e o tipo de retorno na função `fibonacci`. A Figura 6.30 mostra a execução do programa, que exhibe os valores de Fibonacci para vários números.

```

1 // Figura 6.30: fig06_30.cpp
2 // Testando a função fibonacci recursiva.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 unsigned long fibonacci(unsigned long); // protótipo de função
9
10 int main()
11 {
12 // calcula os valores de fibonacci de 0 a 10
13 for (int counter = 0; counter <= 10; counter++)
14 cout << "fibonacci(" << counter << ") = "
15 << fibonacci(counter) << endl;
16
17 // exibe valores fibonacci mais altos
18 cout << "fibonacci(20) = " << fibonacci(20) << endl;
19 cout << "fibonacci(30) = " << fibonacci(30) << endl;
20 cout << "fibonacci(35) = " << fibonacci(35) << endl;
21 return 0; // indica terminação bem-sucedida
22 } // fim de main
23
24 // função fibonacci recursiva
25 unsigned long fibonacci(unsigned long number)
26 {
27 if ((number == 0) || (number == 1)) // casos básicos
28 return number;
29 else // passo de recursão
30 return fibonacci(number - 1) + fibonacci(number - 2);
31 } // fim da função fibonacci

```

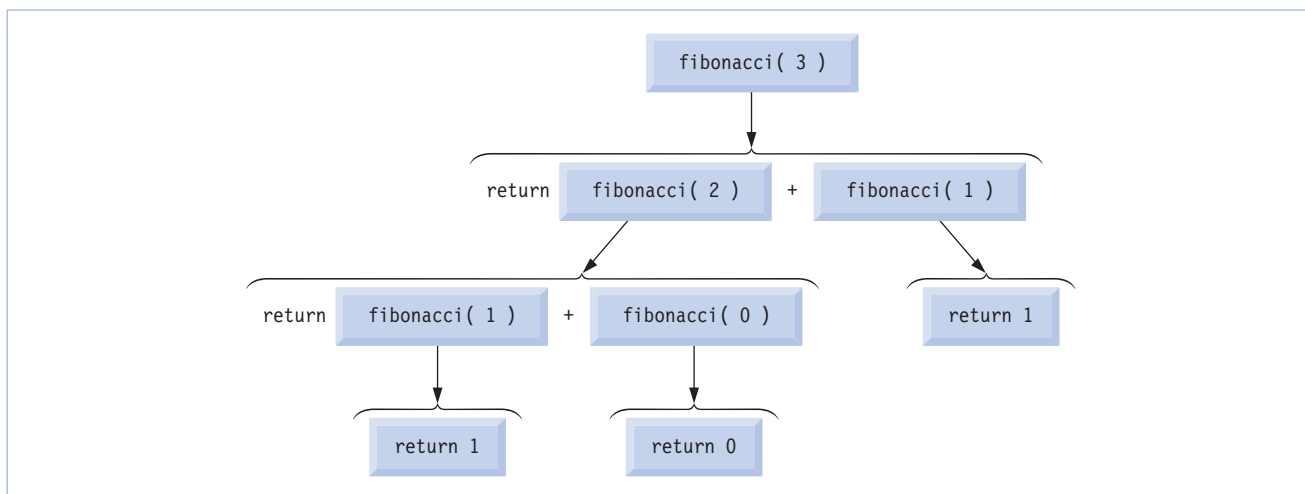
```

fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(2) = 1
fibonacci(3) = 2
fibonacci(4) = 3
fibonacci(5) = 5
fibonacci(6) = 8
fibonacci(7) = 13
fibonacci(8) = 21
fibonacci(9) = 34
fibonacci(10) = 55
fibonacci(20) = 6765
fibonacci(30) = 832040
fibonacci(35) = 9227465

```

**Figura 6.30** Demonstrando a função fibonacci.

O aplicativo inicia com uma instrução `for` que calcula e exibe os valores de Fibonacci para os inteiros 0–10 e é seguida por três chamadas para calcular os valores de Fibonacci dos inteiros 20, 30 e 35 (linhas 18–20). As chamadas a `fibonacci` (linhas 15, 18, 19 e 20) provenientes de `main` não são recursivas, mas as chamadas de `fibonacci` da linha 30 são recursivas. Toda vez que o programa invoca `fibonacci` (linhas 25–31), a função testa imediatamente o caso básico para determinar se `number` é igual a 0 ou 1 (linha 27). Se isso for verdadeiro, a linha 28 retorna `number`. Curiosamente, se `number` for maior que 1, o passo de recursão (linha 30) gera duas chamadas recursivas, cada uma para um problema ligeiramente menor do que a chamada original a `fibonacci`. A Figura 6.31 mostra como a função `fibonacci` avaliaria `fibonacci( 3 )`.



**Figura 6.31** Conjunto de chamadas recursivas à função fibonacci.

Essa figura levanta algumas questões interessantes sobre a ordem em que compiladores C++ avaliarão os operandos dos operadores. Essa é uma questão separada da ordem em que os operadores são aplicados aos seus operandos, a saber, a ordem ditada pelas regras de precedência e associatividade de operadores. A Figura 6.31 mostra que avaliar `fibonacci(3)` produz duas chamadas recursivas, a saber, `fibonacci(2)` e `fibonacci(1)`. Mas em que ordem essas chamadas são feitas? A maioria dos programadores simplesmente pressupõe que os operandos são avaliados da esquerda para a direita. A linguagem C++ não especifica a ordem em que os operandos da maioria dos operadores (inclusive `+`) devem ser avaliados. Portanto, o programador não deve fazer nenhuma suposição sobre a ordem em que essas chamadas executam. As chamadas poderiam de fato executar `fibonacci(2)` primeiro e, então, `fibonacci(1)`, ou poderiam executar na ordem inversa: `fibonacci(1)`, em seguida, `fibonacci(2)`. Nesse programa e na maioria dos outros, revela-se que o resultado final seria o mesmo. Entretanto, em alguns programas a avaliação de um operando pode ter **efeitos colaterais** (alterações nos valores dos dados) que poderiam afetar o resultado final da expressão.

A linguagem C++ especifica a ordem de avaliação dos operandos de apenas quatro operadores — a saber, `&&`, `||`, vírgula `,` e `?:`. Os três primeiros são operadores binários cujos dois operandos são garantidamente avaliados da esquerda para a direita. O último operador é o único operador ternário do C++. Seu operando mais à esquerda sempre é avaliado primeiro; se for avaliado como não-zero (verdadeiro), o operando do meio é avaliado em seguida e o último operando é ignorado; se o operando mais à esquerda for avaliado como zero (falso), o terceiro operando é avaliado em seguida e o operando do meio é ignorado.



### Erro comum de programação 6.25

*Escrever programas que dependem da ordem de avaliação dos operandos de operadores diferentes dos operadores `&&`, `||`, `?:` e vírgula `,` pode levar a erros de lógica.*



### Dica de portabilidade 6.3

*Os programas que dependem da ordem de avaliação dos operandos de operadores diferentes dos operadores `&&`, `||`, `?:` e vírgula `,` podem funcionar diferentemente em sistemas com compiladores diferentes.*

Uma palavra de cautela está em ordem sobre programas recursivos como o que utilizamos aqui para gerar números de Fibonacci. Cada nível de recursão na função `fibonacci` tem o efeito de duplicar o número de chamadas de função; isto é, o número de chamadas recursivas que é requerido para calcular o  $n^{\text{ésimo}}$  número de Fibonacci está na ordem de  $2^n$ . Isso rapidamente foge do controle. Calcular somente o vigésimo número de Fibonacci exigiria um número de chamadas na ordem de  $2^{20}$  ou cerca de um milhão de chamadas, calcular o trigésimo número de Fibonacci exigiria um número de chamadas na ordem de  $2^{30}$  ou cerca de um bilhão de chamadas e assim por diante. Os cientistas da computação se referem a isso como **complexidade exponencial**. Os problemas dessa natureza humilham até os computadores mais poderosos do mundo! Questões de complexidade em geral, e de complexidade exponencial em particular, são discutidas em detalhes em cursos de nível superior de ciência da computação geralmente chamados de ‘Algoritmos’.



### Dica de desempenho 6.8

*Evite programas recursivos no estilo de Fibonacci que resultam em uma ‘explosão’ exponencial de chamadas.*



## 6.2.1 Recursão versus iteração

Nas duas seções anteriores, estudamos duas funções que podem ser facilmente implementadas recursiva ou iterativamente. Esta seção compara as duas abordagens e discute por que o programador poderia escolher uma abordagem à outra em uma situação particular.

Tanto iteração como recursão se baseiam em uma estrutura de controle: a iteração utiliza uma estrutura de repetição; a recursão utiliza uma estrutura de seleção. Ambas envolvem repetição: a iteração utiliza explicitamente uma estrutura de repetição; a recursão alcança repetição por chamadas de função repetidas. Iteração e recursão envolvem um teste de terminação: a iteração termina quando a condição de continuação do loop falha; a recursão termina quando um caso básico é reconhecido. A iteração com repetição controlada por contador e a recursão gradualmente se aproximam do término: a iteração modifica um contador até que o contador assume um valor que faz a condição de continuação do loop falhar; a recursão produz versões mais simples do problema original até que o caso básico seja alcançado. Tanto uma como outra podem ocorrer infinitamente: um loop infinito ocorre com a iteração se o teste de continuação do loop nunca se tornar falso; a recursão infinita ocorre se o passo de recursão não reduz o problema durante cada chamada recursiva de uma maneira que convirja para o caso básico.

Para ilustrar as diferenças entre iteração e recursão, examinemos uma solução iterativa do problema fatorial (Figura 6.32). Observe que uma instrução de repetição é utilizada (linhas 28–29 da Figura 6.32) em vez da instrução de seleção da solução recursiva (linhas 25–28 da Figura 6.29). Observe que as duas soluções utilizam um teste de terminação. Na solução recursiva, a linha 25 testa quanto ao caso básico. Na solução iterativa, a linha 28 testa a condição de continuação do loop — se o teste falhar, o loop termina. Por fim, observe que em vez de produzir a versão mais simples do problema original, a solução iterativa utiliza um contador que é modificado até a condição de continuação do loop tornar-se falsa.

A recursão tem muitos pontos negativos. Ela invoca repetidamente o mecanismo, e conseqüentemente o overhead das chamadas de função. Isso pode ter um alto preço tanto em tempo de processador como em espaço de memória. Cada chamada recursiva faz com que outra cópia da função (na realidade, somente as variáveis da função) seja criada; isso pode consumir memória considerável. A iteração normalmente ocorre dentro de uma função, então o overhead das chamadas de função repetidas e a atribuição extra de memória são omitidos. Então, por que escolher recursão?



### Observação de engenharia de software 6.18

*Qualquer problema que pode ser resolvido recursivamente também pode ser resolvido iterativamente (não recursivamente). Uma abordagem recursiva normalmente é escolhida preferencialmente a uma abordagem iterativa quando a abordagem recursiva espelha mais naturalmente o problema e resulta em um programa que é mais fácil de entender e depurar. Outra razão de escolher uma solução recursiva é que uma solução iterativa não é evidente.*



### Dica de desempenho 6.9

*Evite utilizar recursão em situações de desempenho. Chamadas recursivas levam tempo e consomem memória adicional.*



### Erro comum de programação 6.26

*Ter acidentalmente uma função não recursiva chamando a si própria, direta ou indiretamente (por outra função), é um erro de lógica.*

A maioria dos manuais de programação introduz recursão muito mais tarde do que fizemos aqui. Mas nós acreditamos que a recursão é um tópico complexo e suficientemente rico e é melhor introduzi-lo mais cedo e espalhar os exemplos pelo restante do texto. A Figura 6.33 resume os exemplos e exercícios de recursão no texto.

```

1 // Figura 6.32: fig06_32.cpp
2 // Testando a função fatorial iterativa.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 unsigned long factorial(unsigned long); // protótipo de função
11
```

**Figura 6.32** Solução fatorial iterativa.

(continua)

```

12 int main()
13 {
14 // calcula o fatorial de 0 a 10
15 for (int counter = 0; counter <= 10; counter++)
16 cout << setw(2) << counter << "! = " << factorial(counter)
17 << endl;
18
19 return 0;
20 } // fim de main
21
22 // função fatorial iterativa
23 unsigned long factorial(unsigned long number)
24 {
25 unsigned long result = 1;
26
27 // declaração iterativa da função fatorial
28 for (unsigned long i = number; i >= 1; i--)
29 result *= i;
30
31 return result;
32 } // fim da função fatorial

```

```

0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800

```

Figura 6.32 Solução fatorial iterativa.

(continuação)

| Localização no texto |                         | Exemplos e exercícios de recursão             |
|----------------------|-------------------------|-----------------------------------------------|
| Capítulo 6           | Seção 6.19, Figura 6.29 | Função fatorial                               |
|                      | Seção 6.19, Figura 6.30 | Função de Fibonacci                           |
|                      | Exercício 6.7           | Soma de dois inteiros                         |
|                      | Exercício 6.40          | Elevando um inteiro a uma potência inteira    |
|                      | Exercício 6.42          | Torres de Hanói                               |
|                      | Exercício 6.44          | Visualizando a recursão                       |
|                      | Exercício 6.45          | Máximo divisor comum recursivo                |
|                      | Exercícios 6.50 e 6.51  | Exercício Mistério “O que esse programa faz?” |
| Capítulo 7           | Exercício 7.18          | Exercício Mistério “O que esse programa faz?” |
|                      | Exercício 7.21          | Exercício Mistério “O que esse programa faz?” |
|                      | Exercício 7.31          | Classificação por seleção                     |

Figura 6.33 Resumo de exemplos e exercícios de recursão no texto.

(continua)

| Localização no texto | Exemplos e exercícios de recursão |
|----------------------|-----------------------------------|
| Capítulo 8           | Exercício 7.32                    |
|                      | Exercício 7.33                    |
|                      | Exercício 7.34                    |
|                      | Exercício 7.35                    |
|                      | Exercício 7.36                    |
|                      | Exercício 7.37                    |
|                      | Exercício 7.38                    |
|                      | Exercício 8.24                    |
|                      | Exercício 8.25                    |
|                      | Exercício 8.26                    |
| Capítulo 20          | Exercício 8.27                    |
|                      | Seção 20.3.3, figuras 20.5–20.7   |
|                      | Exercício 20.8                    |
|                      | Exercício 20.9                    |
| Capítulo 21          | Exercício 20.10                   |
|                      | Seção 21.7, figuras 21.20–21.22   |
|                      | Seção 21.7, figuras 21.20–21.22   |
|                      | Seção 21.7, figuras 21.20–21.22   |
|                      | Seção 21.7, figuras 21.20–21.22   |
|                      | Exercício 21.20                   |
|                      | Exercício 21.21                   |
|                      | Exercício 21.22                   |
|                      | Exercício 21.25                   |

Figura 6.33 Resumo de exemplos e exercícios de recursão no texto.

(continuação)

## 6.22 Estudo de caso de engenharia de software: identificando operações de classe no sistema ATM (opcional)

Nas seções de “Estudo de caso de engenharia de software” no final dos capítulos 3, 4 e 5, seguimos os primeiros passos do projeto orientado a objetos do nosso sistema ATM. No Capítulo 3, identificamos as classes que precisaremos implementar e criamos nosso primeiro diagrama de classes. No Capítulo 4, descrevemos alguns atributos das nossas classes. No Capítulo 5, examinamos estados dos objetos e transições de estado e atividades dos objetos modelados. Nesta seção, determinamos algumas operações de classe (ou comportamentos) necessárias para implementar o sistema ATM.

### Identificando operações

Uma operação é um serviço que os objetos de uma classe fornecem aos clientes da classe. Pense nas operações de alguns objetos do mundo real. As operações de um rádio incluem configurar sua estação e volume (em geral invocadas por uma pessoa que ajusta os controles do rádio). As operações de um carro incluem acelerar (invocada pelo motorista ao pressionar pedal do acelerador), desacelerar (invocada pelo motorista que pressiona o pedal do freio ou solta o pedal do acelerador), mudar de direção e trocar de marchas. Os objetos de software também podem oferecer operações — por exemplo, um objeto de um software gráfico poderia oferecer operações para desenhar um círculo, uma linha, um quadrado etc. Um objeto de um software de planilha poderia oferecer operações como imprimir a planilha, somar os elementos em uma linha ou coluna e diagramar informações na planilha, como um gráfico de barras ou gráfico de torta.

Podemos derivar várias operações de cada classe examinando os verbos e frases com verbos-chave no documento de requisitos. Então relacionamos cada um desses aspectos a classes particulares no nosso sistema (Figura 6.34). As frases com verbos na Figura 6.34 ajudam a determinar as operações de cada classe.

### Modelando operações

Para identificar operações, examinamos as frases com verbos listadas para cada classe na Figura 6.34. A frase ‘executa transações financeiras’ associada à classe ATM implica que a classe ATM instrui as transações a serem executadas. Portanto, as classes *BalanceInquiry*, *Withdrawal* e *Deposit* precisam de uma operação para fornecer esse serviço ao ATM. Colocamos essa operação (que identificamos como *execute*) no terceiro compartimento das três classes de transação no diagrama de classes atualizado da Figura 6.35. Durante uma sessão no ATM, o objeto ATM invocará a operação *execute* de cada objeto de transação para instruí-lo a executar.

A UML representa operações (implementadas como funções-membro em C++) listando o nome da operação, seguido por uma lista separada por vírgulas de parâmetros entre parênteses, dois-pontos e o tipo de retorno:

*nomeDaOperação ( parâmetro1, parâmetro2, ..., parâmetroN ) : tipo de retorno*

Cada parâmetro na lista separada por vírgulas de parâmetros consiste em um nome de parâmetro, seguido por dois-pontos e o tipo de parâmetro:

*nomeDoParâmetro : tipoDoParâmetro*

Agora, não listamos os parâmetros das nossas operações — identificaremos e modelaremos os parâmetros de algumas operações mais adiante. Para algumas operações, ainda não conhecemos os tipos de retorno, portanto também iremos omiti-los do diagrama. Essas omissões são perfeitamente normais nesse ponto. À medida que o nosso projeto e implementação avançarem, adicionaremos os tipos de retorno remanescentes.

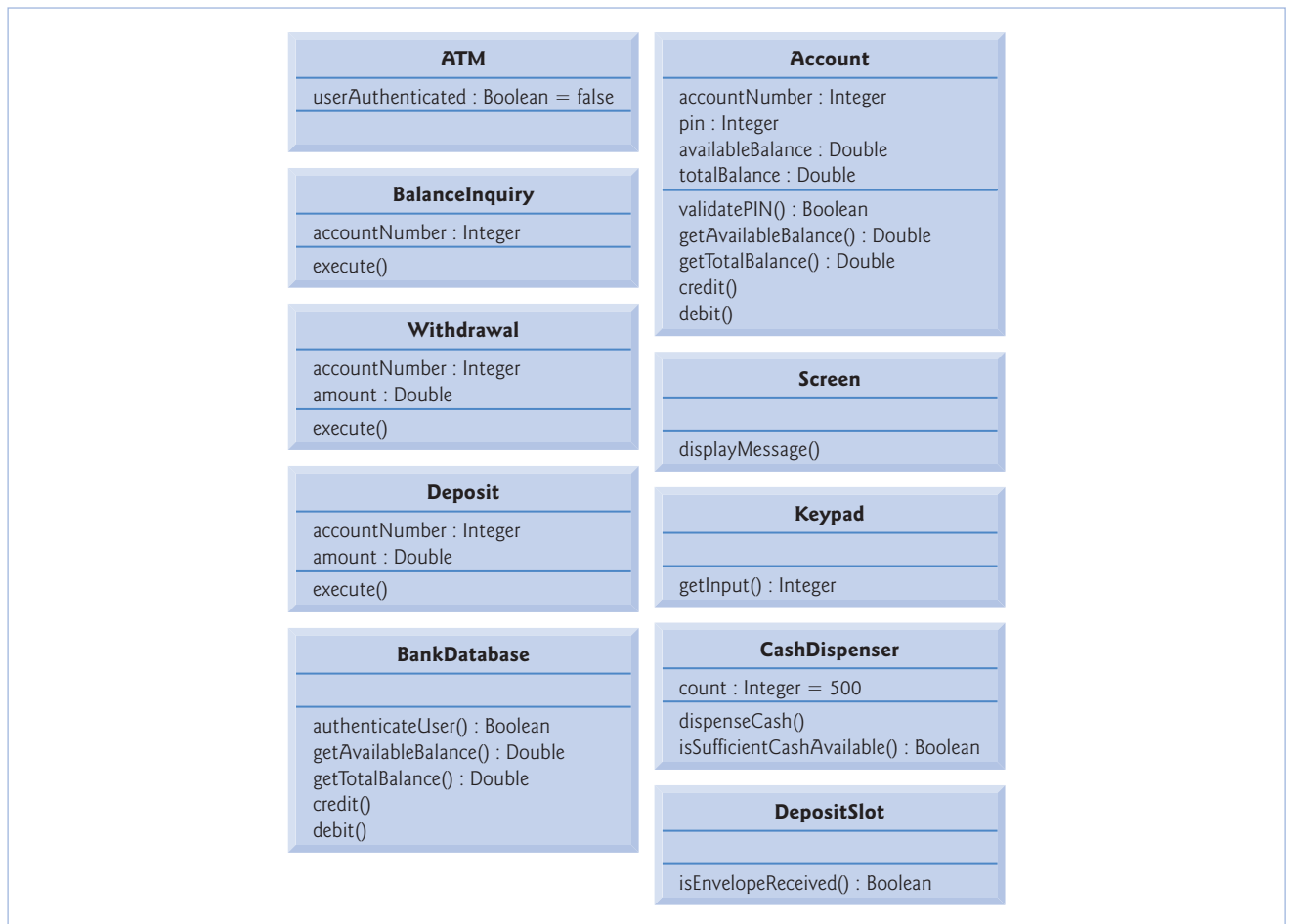
### Operações da classe *BankDatabase* e da classe *Account*

A Figura 6.34 lista a frase ‘autentica um usuário’ ao lado da classe *BankDatabase* — o banco de dados é o objeto que contém informações sobre uma conta necessárias para determinar se o número da conta e o PIN inserido por um usuário correspondem àqueles de uma conta mantida pelo banco. Portanto, a classe *BankDatabase* precisa de uma operação que forneça um serviço de autenticação ao ATM. Colocamos a operação *authenticateUser* no terceiro compartimento da classe *BankDatabase* (Figura 6.35). Entretanto, um objeto da classe *Account*, não da classe *BankDatabase*, armazena o número da conta e o PIN que devem ser acessados para autenticar um usuário; dessa forma, a classe *Account* deve fornecer um serviço para validar um PIN, obtido por meio da entrada do usuário, contra um PIN armazenado em um objeto *Account*. Portanto, adicionamos uma operação *validatePIN* à classe *Account*. Observe que especificamos um tipo de retorno *Boolean* para as operações *authenticateUser* e *validatePIN*. Cada operação retorna um valor que indica que a operação foi bem-sucedida na realização de sua tarefa (isto é, um valor de retorno de *true*) ou que não foi (isto é, um valor de retorno de *false*).

A Figura 6.34 lista várias frases com verbos adicionais da classe *BankDatabase*: ‘recupera um saldo em conta’, ‘credita uma quantia depositada em uma conta’ e ‘debita uma quantia retirada de uma conta’. Como ocorre com ‘autentica um usuário’, essas frases restantes se referem aos serviços que o banco de dados deve fornecer ao ATM, porque o banco de dados armazena todos os dados de uma conta

| Classe         | Verbos e frases com verbos                                                                                                              |
|----------------|-----------------------------------------------------------------------------------------------------------------------------------------|
| ATM            | executa transações financeiras                                                                                                          |
| BalanceInquiry | [nenhuma no documento de requisitos]                                                                                                    |
| Withdrawal     | [nenhuma no documento de requisitos]                                                                                                    |
| Deposit        | [nenhuma no documento de requisitos]                                                                                                    |
| BankDatabase   | autentica um usuário, recupera um saldo em conta, credita uma quantia depositada em uma conta, debita uma quantia retirada de uma conta |
| Account        | recupera um saldo em conta, credita uma quantia depositada em uma conta, debita uma quantia retirada de uma conta                       |
| Screen         | exibe uma mensagem para o usuário                                                                                                       |
| Keypad         | recebe entrada numérica do usuário                                                                                                      |
| CashDispenser  | fornece o dinheiro, indica se contém dinheiro suficiente para satisfazer uma solicitação de saque                                       |
| DepositSlot    | recebe um envelope de depósito                                                                                                          |

**Figura 6.34** Os verbos e frases com verbo para cada classe no sistema ATM.



**Figura 6.35** Classes no sistema ATM com atributos e operações.

utilizada para autenticar um usuário e realizar as transações no ATM. Entretanto, objetos da classe `Account` na verdade realizam as operações às quais essas frases se referem. Portanto, atribuímos uma operação à classe `BankDatabase` e à classe `Account` para que elas correspondam a cada uma dessas frases. Lembre-se, a partir do que foi discutido na Seção 3.11, de que, como conta bancária contém informações sigilosas, não permitimos que o ATM acesse contas diretamente. O banco de dados atua como um intermediário entre o ATM e os dados da conta, evitando assim acesso não autorizado. Como veremos na Seção 7.12, a classe `ATM` invoca as operações da classe `BankDatabase`, cada uma das quais por sua vez invoca a operação com o mesmo nome na classe `Account`.

A frase ‘recupera um saldo em conta’ sugere que as classes `BankDatabase` e `Account` precisam de uma operação `getBalance`. Entretanto, lembre-se de que criamos dois atributos na classe `Account` a fim de representar um saldo — `availableBalance` e `totalBalance`. Uma consulta de saldo requer acesso aos dois atributos de saldo para poder exibi-los ao usuário, mas um saque precisa verificar somente o valor de `availableBalance`. Para permitir que objetos no sistema obtenham cada atributo de saldo individualmente, adicionamos operações `getAvailableBalance` e `getTotalBalance` ao terceiro compartimento das classes `BankDatabase` e `Account` (Figura 6.35). Especificamos um tipo de retorno de `Double` para cada uma dessas operações porque os atributos de saldo que eles recuperam são do tipo `Double`.

As frases ‘credita uma quantia depositada em uma conta’ e ‘debita uma quantia retirada de uma conta’ indicam que as classes `BankDatabase` e `Account` devem realizar operações para atualizar uma conta durante um depósito e um saque, respectivamente. Portanto, atribuímos as operações `credit` e `debit` às classes `BankDatabase` e `Account`. Lembre-se de que creditar em uma conta (como em um depósito) só adiciona uma quantia monetária ao atributo `totalBalance`. Debitar de uma conta (como em um saque), por outro lado, subtrai a quantia dos dois atributos de saldo. Ocultamos esses detalhes de implementação dentro da classe `Account`. Esse é um bom exemplo do encapsulamento e ocultamento de informações.

Se isso fosse um sistema ATM real, as classes `BankDatabase` e `Account` também forneceriam um conjunto de operações para permitir que outro sistema de operações bancárias atualizasse um saldo na conta do usuário depois de uma confirmação ou rejeição de todo ou parte de um depósito. A operação `confirmDepositAmount`, por exemplo, adicionaria uma quantia monetária ao atributo `availableBalance`, tornando assim os fundos depositados disponíveis para saque. A operação `rejectDepositAmount` subtrairia uma quantia monetária do atributo `totalBalance` para indicar que uma quantia especificada, que foi recentemente depositada por meio do ATM e adicionada ao

`totalBalance`, não foi encontrada no envelope de depósito. O banco invocaria essa operação depois de determinar que o usuário não incluiu a quantia monetária correta ou que um cheque não foi compensado (isto é, ele ‘voltou’). Adicionar essas operações torna nosso sistema mais completo, porém não as incluímos nos nossos diagramas de classes nem na nossa implementação porque elas estão além do escopo desse estudo de caso.

### Operações da classe **Screen**

A classe `Screen` ‘exibe uma mensagem para o usuário’ em vários momentos em uma sessão no ATM. Toda a saída visual ocorre por meio da tela do ATM. O documento de requisitos descreve muitos tipos de mensagens (por exemplo, uma mensagem de boas-vindas, uma mensagem de erro, uma mensagem de agradecimento) que a tela exibe para o usuário. O documento de requisitos também indica que a tela exibe prompts e menus para o usuário. Entretanto, um prompt na verdade é apenas uma mensagem que descreve o que o usuário deve inserir em seguida e um menu é essencialmente um tipo de prompt que consiste em uma série de mensagens (isto é, opções de menu) exibida consecutivamente. Portanto, em vez de atribuir a classe `Screen` a uma operação individual a fim de exibir cada tipo de mensagem, prompt e menu, simplesmente criamos uma operação que possa exibir qualquer mensagem especificada por um parâmetro. Colocamos essa operação (`displayMessage`) no terceiro compartimento da classe `Screen` no nosso diagrama de classes (Figura 6.35). Observe que, nesse momento, não nos preocupamos com o parâmetro dessa operação — modelaremos o parâmetro mais tarde nesta seção.

### Operações da classe **Keypad**

Na frase ‘recebe entrada numérica do usuário’ listada pela classe `Keypad` na Figura 6.34, concluímos que a classe `Keypad` deve realizar uma operação `getInput`. Como o teclado do ATM, diferentemente de um teclado de computador, contém somente os números de 0 a 9, especificamos que essa operação retorna um valor inteiro. Lembre-se de que no documento de requisitos, em diferentes situações, talvez seja necessário que o usuário insira um tipo diferente de número (por exemplo, um número de conta, um PIN, o número de uma opção de menu, uma quantia de depósito como um número de centavos). A classe `Keypad` simplesmente obtém um valor numérico para um cliente da classe — ela não determina se o valor atende quaisquer critérios específicos. Toda classe que utiliza essa operação deve verificar se o usuário insere números apropriados e, se não inserir, deve exibir mensagens de erro por meio da classe `Screen`. [Nota: Quando implementamos o sistema, simulamos o teclado do ATM com um teclado de computador e, por simplicidade, supomos que o usuário não irá inserir uma entrada não-numérica utilizando as teclas no teclado de computador que não aparecem no teclado do ATM. Mais adiante no livro, você aprenderá a examinar entradas para determinar se elas são de tipos particulares.]

### Operações da classe **CashDispenser** e classe **DepositSlot**

A Figura 6.34 lista ‘fornece dinheiro’ para a classe `CashDispenser`. Portanto, criamos a operação `dispenseCash` e a listamos sob a classe `CashDispenser` na Figura 6.35. A classe `CashDispenser` também ‘indica se contém dinheiro suficiente para satisfazer uma solicitação de saque’. Portanto, incluímos `isSufficientCashAvailable`, uma operação que retorna um valor do tipo `Boolean`, da UML, na classe `CashDispenser`. A Figura 6.34 também lista ‘recebe um envelope de depósito’ para a classe `DepositSlot`. A abertura para depósito deve indicar se recebeu um envelope, portanto colocamos uma operação `isEnvelopeReceived`, que retorna um valor `Boolean`, no terceiro compartimento da classe `DepositSlot`. [Nota: Um hardware real da abertura para depósito provavelmente enviaria um sinal para o ATM para indicar que um envelope foi recebido. Entretanto, simulamos esse comportamento com uma operação na classe `DepositSlot` para que a classe ATM possa ser invocada a fim de descobrir se a abertura para depósito recebeu um envelope.]

### Operações da classe **ATM**

Não listamos nenhuma operação para a classe ATM nesse momento. Ainda não estamos cientes de nenhum serviço que a classe ATM fornece para outras classes no sistema. Entretanto, ao implementarmos o sistema com código C++, operações dessa classe, e operações adicionais das outras classes no sistema, podem surgir.

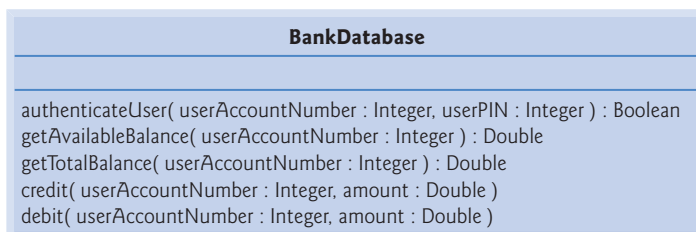
### Identificando e modelando parâmetros de operação

Até esse momento, não nos preocupamos com os parâmetros das nossas operações — tentamos apenas obter um entendimento básico sobre as operações de cada classe. Agora, vamos examinar mais detalhadamente alguns parâmetros de operação. Identificamos os parâmetros de uma operação examinando quais dados a operação requer para realizar sua tarefa atribuída.

Considere a operação `authenticateUser` da classe `BankDatabase`. Para autenticar um usuário, essa operação deve conhecer o número de conta e o PIN fornecido pelo usuário. Assim, especificamos que a operação `authenticateUser` recebe parâmetros inteiros `userAccountNumber` e `userPIN`, os quais a operação deve comparar com o número de conta e o PIN de um objeto `Account` no banco de dados. Prefixamos esses nomes de parâmetros com ‘user’ para evitar confusão entre os nomes de parâmetro da operação e os nomes dos atributos que pertencem à classe `Account`. Listamos esses parâmetros no diagrama de classes na Figura 6.36, que modela somente a classe `BankDatabase`. [Nota: É perfeitamente normal modelar somente uma classe em um diagrama de classes. Nesse caso, estamos mais preocupados em examinar os parâmetros dessa única classe em particular, portanto omitimos as outras classes. Nos diagramas de classes posteriores neste estudo de caso, em que parâmetros não são mais o foco, omitimos esses parâmetros para economizar espaço. Mas não se esqueça de que as operações listadas nesses diagramas ainda contêm parâmetros.]

Lembre-se de que a UML modela cada parâmetro em uma lista de parâmetros separada por vírgulas da operação e listando o nome do parâmetro seguido por um caractere de dois-pontos e pelo tipo de parâmetro (na notação da UML). Assim, a Figura 6.36 especifica que a operação `authenticateUser` recebe dois parâmetros — `userAccountNumber` e `userPIN`, ambos do tipo `Integer`. Quando implementarmos o sistema em C++, representaremos esses parâmetros com os valores `int`.





**Figura 6.36** A classe `BankDatabase` com parâmetros de operação.

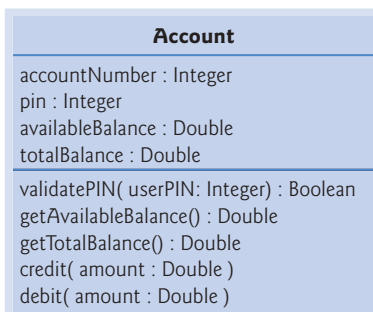
A classe `BankDatabase` com operações `getAvailableBalance`, `getTotalBalance`, `credit` e `debit` também requer um parâmetro `userAccountNumber` para identificar a conta à qual o banco de dados deve aplicar as operações, portanto incluímos esses parâmetros ao diagrama de classes da Figura 6.36. Além disso, as operações `credit` e `debit` requerem um parâmetro `Double amount` para especificar a quantia monetária a ser creditada ou debitada, respectivamente.

O diagrama de classes na Figura 6.37 modela os parâmetros das operações da classe `Account`. A operação `validatePIN` requer somente um parâmetro `userPIN`, que contém o PIN especificado pelo usuário a ser comparado com o PIN associado com a conta. Como ocorre com suas contrapartes na classe `BankDatabase`, as operações `credit` e `debit` na classe `Account` requerem um parâmetro `Double amount` que indica a quantia monetária envolvida na operação. As operações `getAvailableBalance` e `getTotalBalance` na classe `Account` não requerem nenhum dado adicional para realizar suas tarefas. Observe que as operações da classe `Account` não requerem um parâmetro de número de conta — cada uma dessas operações pode ser invocada somente em um objeto `Account` específico, portanto, é desnecessário incluir um parâmetro para especificar um `Account`.

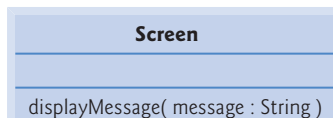
A Figura 6.38 modela a classe `Screen` com um parâmetro especificado para a operação `displayMessage`. Essa operação requer somente um parâmetro `String message` que indica o texto a ser exibido. Lembre-se de que os tipos de parâmetros listados nos nossos diagramas de classes estão em notação UML, portanto o tipo `String` listado na Figura 6.38 refere-se ao tipo UML. Quando implementarmos o sistema em C++, iremos, de fato, utilizar um objeto `string` C++ para representar esse parâmetro.

O diagrama de classes na Figura 6.39 especifica que a operação `dispenseCash` da classe `CashDispenser` recebe um parâmetro `Double amount` para indicar a quantia monetária (em dólares) a ser entregue. A operação `isSufficientCashAvailable` também recebe um parâmetro `Double amount` para indicar a quantia monetária em questão.

Observe que não discutimos os parâmetros para a operação `execute` das classes `BalanceInquiry`, `Withdrawal` e `Deposit`, a operação `getInput` da classe `Keypad` e a operação `isEnvelopeReceived` da classe `DepositSlot`. Nesta fase do nosso processo de projeto, não podemos determinar se essas operações exigem dados adicionais para realizar suas tarefas, assim deixamos suas listas de parâmetro vazias. À medida que avançamos pelo estudo de caso, podemos decidir adicionar parâmetros a essas operações.

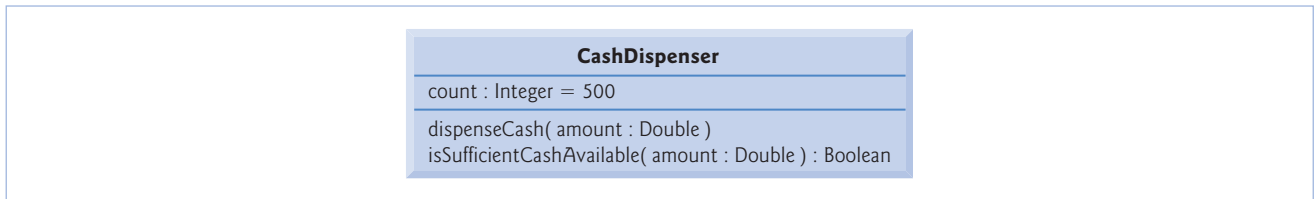


**Figura 6.37** A classe `Account` com parâmetros de operação.



**Figura 6.38** A classe `Screen` com parâmetros de operação.





**Figura 6.39** A classe CashDispenser com parâmetros de operação.

Nesta seção, determinamos várias operações realizadas pelas classes no sistema ATM. Identificamos os parâmetros e os tipos de retorno de algumas operações. À medida que prosseguimos pelo nosso processo de projeto, o número de operações que pertencem a cada classe talvez varie — poderíamos descobrir que novas operações são necessárias ou que algumas operações atuais são desnecessárias — e poderíamos determinar que algumas das nossas operações de classe precisam de parâmetros adicionais e tipos de retorno diferentes.

### Exercícios de revisão do estudo de caso de engenharia de software

- 6.1** Qual das seguintes alternativas não é um comportamento?
- ler dados a partir de um arquivo
  - imprimir a saída
  - gerar saída de texto
  - obter a entrada do usuário
- 6.2** Se você fosse adicionar ao sistema ATM uma operação que retornasse o atributo amount da classe Withdrawal, como e onde você especificaria essa operação no diagrama de classes da Figura 6.35?
- 6.3** Descreva o significado da listagem da operação a seguir que poderia aparecer em um diagrama de classes em um projeto orientado a objetos de uma calculadora:
- ```
add( x : Integer, y : Integer ) : Integer
```

Respostas aos exercícios de revisão do estudo de caso de engenharia de software

- 6.1** c.
- 6.2** Para especificar uma operação que recupera o atributo amount da classe Withdrawal, a operação a seguir seria colocada no compartimento de operações (isto é, terceiro) da classe Withdrawal:
- ```
getAmount() : Double
```
- 6.3** Essa é uma operação chamada add que aceita inteiros x e y como parâmetros e retorna um valor inteiro.

## 6.23 Síntese

Neste capítulo, você aprendeu mais sobre os detalhes das declarações de função. As funções têm partes diferentes, como o protótipo, a assinatura, o cabeçalho e o corpo de função. Você aprendeu sobre a coerção de argumento, isto é, forçar argumentos para os tipos apropriados especificados pelas declarações de parâmetro de uma função. Demonstramos como utilizar funções rand e srand para gerar conjuntos de números aleatórios que podem ser utilizados para simulações. Você também aprendeu sobre o escopo de variáveis ou a parte de um programa em que um identificador pode ser utilizado. Duas maneiras diferentes de passar argumentos para funções foram discutidas — passagem por valor e por referência. Na passagem por referência, as referências são utilizadas como um alias para uma variável. Você aprendeu que múltiplas funções em uma classe podem ser sobrecarregadas fornecendo funções com o mesmo nome e assinaturas diferentes. Essas funções podem ser utilizadas para realizar as mesmas tarefas, ou tarefas semelhantes, utilizando tipos diferentes ou números diferentes de parâmetros. Depois demonstramos uma maneira mais simples de sobrecarregar funções para utilizar templates de função, onde uma função é definida uma vez, mas pode ser utilizada para vários tipos diferentes. Então introduzimos o conceito de recursão, onde uma função chama a si mesma para resolver um problema.

No Capítulo 7, você aprenderá a manter listas e tabelas de dados em arrays. Você verá uma implementação baseada em array mais elegante do aplicativo de lançamento de dados e duas versões aprimoradas de nosso estudo de caso GradeBook que você acompanhou nos capítulos 3–5 e que utilizarão arrays para armazenar as notas reais inseridas.

### Resumo

- A experiência mostra que a melhor maneira de desenvolver e manter um programa grande é construí-lo a partir de partes pequenas e simples, ou módulos. Essa técnica se chama dividir para conquistar.

- Em geral, os programas C++ são escritos combinando novas funções e classes que o programador escreve com funções ‘pré-empacotadas’ e classes disponíveis na C++ Standard Library.
- As funções permitem ao programador modularizar um programa separando suas tarefas em unidades autocontidas.
- As instruções no corpo das funções são escritas apenas uma vez, talvez reutilizadas a partir de diversas localizações em um programa, e são ocultadas de outras funções.
- O compilador se refere ao protótipo de função para verificar se as chamadas para uma função contêm o número e os tipos de argumentos corretos, se os tipos dos argumentos estão na ordem correta e se o valor retornado pela função pode ser utilizado corretamente na expressão que chamou a função.
- Há três maneiras de retornar o controle para o ponto em que uma função foi invocada. Se a função não retornar um resultado, o controle retornará quando o programa alcançar a chave direita de fechamento da função, ou pela execução da instrução

```
return;
```

Se a função retorna um resultado, a instrução

```
return expressão;
```

avalia *expressão* e retorna o valor de *expressão* para o chamador.

- Um protótipo de função informa ao compilador o nome de uma função, os tipos de dados retornados por essa função, o número de parâmetros que a função espera receber, os tipos desses parâmetros e a ordem em que os parâmetros desses tipos são esperados.
- A parte de um protótipo de função que inclui o nome da função e os tipos de seus argumentos é chamada assinatura de função ou simplesmente assinatura.
- Um recurso importante dos protótipos de função é a coerção de argumento — isto é, forçar argumentos para os tipos apropriados especificados pelas declarações de parâmetro.
- Os valores de argumento que não correspondem precisamente aos tipos de parâmetro no protótipo de função podem ser convertidos pelo compilador no tipo adequado como especificado pelas regras de promoção do C++. As regras de promoção indicam como converter entre tipos sem perder dados.
- O elemento chance pode ser introduzido em aplicativos de computador utilizando a função `rand` da C++ Standard Library.
- A função `rand` realmente gera números pseudo-aleatórios, ou pseudo-randômicos. Chamar `rand` repetidamente produz uma seqüência de números que parece aleatória. Entretanto, a seqüência repete-se toda vez que o programa executa.
- Uma vez que um programa foi completamente depurado, ele pode ser condicionado a produzir uma seqüência diferente de números aleatórios para cada execução. Isso é chamado aleatorização ou randomização e é realizado com a função `srand` da C++ Standard Library C++.
- A função `srand` aceita um argumento do tipo inteiro `unsigned` e semeia a função `rand` para produzir uma seqüência diferente de números aleatórios para cada execução do programa.
- Os números aleatórios em um intervalo podem ser gerados com

```
número = valorDeDeslocamento + rand() % fatorDeEscala;
```

onde *valorDeDeslocamento* é igual ao primeiro número no intervalo desejado de inteiros consecutivos e *fatorDeEscala* é igual à largura do intervalo desejado de inteiros consecutivos.

- Uma enumeração, introduzida pela palavra-chave `enum` e seguida por um nome de tipo, é um conjunto de constantes do tipo inteiro representadas por identificadores. Os valores dessas constantes enumeradas iniciam em 0, a menos que especificado de outro modo, e incrementa por 1.
- A classe de armazenamento de um identificador determina o período em que esse identificador existe na memória.
- O escopo de um identificador é onde o identificador pode ser referenciado em um programa.
- A linkagem de um identificador determina se um identificador só é conhecido no arquivo de fonte onde ele é declarado ou por múltiplos arquivos que são compilados e, então, linkados.
- As palavras-chave `auto` e `register` são utilizadas para declarar as variáveis da classe de armazenamento automática. Essas variáveis são criadas quando a execução do programa insere o bloco em que elas são definidas, existem enquanto o bloco está ativo e são destruídas quando o programa sai do bloco.
- Somente variáveis locais de uma função podem ser de classe de armazenamento automática.
- O especificador de classe de armazenamento `auto` declara explicitamente variáveis de classe de armazenamento automática. As variáveis locais são da classe de armazenamento automática por padrão, então a palavra-chave `auto` é raramente utilizada.
- As palavras-chave `extern` e `static` declaram identificadores para variáveis da classe de armazenamento estática e para funções. As variáveis de classe de armazenamento estática existem a partir do ponto em que o programa começa a execução e duram até o fim do programa.
- O armazenamento de uma variável de uma classe de armazenamento estático é alocado quando o programa começa a execução. Essa variável é inicializada uma vez quando sua declaração é encontrada. Para funções, o nome da função existe a partir do momento em que o programa começa a executar, assim como ocorre para todas as outras funções.

- Há dois tipos de identificadores com classe de armazenamento estática — os identificadores externos (como as variáveis globais e os nomes de função globais) e as variáveis locais declaradas com o especificador de classe de armazenamento `static`.
- As variáveis globais são criadas colocando-se as declarações de variável fora de qualquer classe ou definição de função. As variáveis globais retêm seus valores por toda a execução do programa. Variáveis globais e funções globais podem ser referenciadas por qualquer função que siga suas declarações ou definições no arquivo-fonte.
- As variáveis locais declaradas com a palavra-chave `static` ainda são conhecidas apenas na função em que são declaradas, mas, ao contrário das variáveis automáticas, as variáveis locais `static` retêm seus valores quando a função retorna para seu chamador. A próxima vez em que a função é chamada, as variáveis locais `static` contêm os valores que tinham quando a função completou pela última vez a execução.
- Um identificador declarado fora de qualquer função ou classe tem escopo de arquivo.
- Os rótulos são os únicos identificadores com escopo de função. Os rótulos podem ser utilizados em qualquer lugar na função em que aparecem, mas não podem ser referenciados fora do corpo da função.
- Os identificadores declarados dentro de um bloco têm escopo de bloco. O escopo de bloco começa na declaração do identificador e termina na chave de fechamento direita (}) do bloco em que o identificador é declarado.
- Os únicos identificadores com escopo de protótipo de função são aqueles utilizados na lista de parâmetros de um protótipo de função.
- As pilhas são conhecidas como estruturas de dados do tipo último a entrar, primeiro a sair (*last-in, first-out* – LIFO) — o último item inserido na pilha é o primeiro item que é removido da pilha.
- Um dos mecanismos mais importantes para ser entendido pelos alunos de ciência da computação é a pilha de chamadas de função (às vezes referida como pilha de execução do programa). Essa estrutura de dados suporta o mecanismo chamada/retorno de função.
- A pilha de chamadas de função também suporta a criação, manutenção e destruição de variáveis automáticas de cada função chamada.
- Toda vez que uma função chama outra função, uma entrada é empurrada sobre a pilha. Essa entrada, chamada de quadro de pilha (*stack frame*) ou registro de ativação, contém o endereço de retorno de que a função chamada precisa para retornar à função chamadora, bem como as variáveis automáticas para a chamada de função.
- O quadro de pilha existe enquanto a função chamada estiver ativa. Quando a função chamada retornar — e não precisar mais de suas variáveis automáticas locais —, seu quadro de pilha é removido da pilha e essas variáveis automáticas locais não são mais conhecidas pelo programa.
- No C++, uma lista vazia de parâmetros é especificada escrevendo `void` ou não escrevendo nada entre parênteses.
- O C++ fornece funções `inline` para ajudar a reduzir o overhead da chamada de função — especialmente para funções pequenas. Colocar o qualificador `inline` antes do tipo de retorno de uma função na definição de função ‘aconselha’ o compilador a gerar uma cópia do código da função no lugar para evitar uma chamada de função.
- Duas maneiras de passar argumentos para funções em muitas linguagens de programação são a passagem por valor e a passagem por referência.
- Quando um argumento é passado por valor, uma cópia do valor do argumento é feita e passada (na pilha de chamadas de função) para a função chamada. As alterações na cópia não afetam o valor da variável original no chamador.
- Com a passagem por referência, o chamador fornece à função chamada a capacidade de acessar os dados do chamador diretamente e modificá-los se a função chamada escolher fazer isso.
- Um parâmetro de referência é um *alias* (um ‘apelido’, pronuncia-se ‘álias’) para seu argumento correspondente em uma chamada de função.
- Para indicar que um parâmetro de função é passado por referência, simplesmente coloque um ‘e comercial’ (&) depois do tipo do parâmetro no protótipo de função; use a mesma convenção ao listar o tipo do parâmetro no cabeçalho de função.
- Uma vez que uma referência é declarada como um alias para outra variável, todas as operações supostamente realizadas no alias (isto é, a referência) são realmente realizadas na variável original. O alias tem simplesmente outro nome para a variável original.
- Não é incomum que um programa invoque uma função repetidamente com o mesmo valor de argumento para um parâmetro particular. Nesses casos, o programador pode especificar que tal parâmetro tem um argumento-padrão, isto é, um valor-padrão a ser passado a esse parâmetro.
- Quando um programa omite um argumento para um parâmetro com um argumento-padrão, o compilador reescreve a chamada de função e insere o valor-padrão desse argumento a ser passado como um argumento para a chamada de função.
- Argumentos-padrão devem ser os argumentos mais à direita (finais) em uma lista de parâmetros da função.
- Os argumentos-padrão devem ser especificados com a primeira ocorrência do nome de função — em geral, no protótipo de função.
- O C++ fornece o operador unário de resolução de escopo (`::`) para acessar uma variável global quando uma variável local do mesmo nome está no escopo.
- O C++ permite que várias funções do mesmo nome sejam definidas, contanto que essas funções tenham diferentes conjuntos de parâmetros. Essa capacidade é chamada de sobrecarga de funções.
- Quando uma função sobrecarregada é chamada, o compilador C++ seleciona a função adequada examinando o número, os tipos e a ordem dos argumentos na chamada.
- As funções sobrecarregadas são distinguidas por suas assinaturas.

- O compilador codifica cada identificador de função com o número e os tipos de seus parâmetros para permitir a linkagem segura para o tipo. A linkagem segura para o tipo garante que a função sobrecarregada adequada seja chamada e que os tipos dos argumentos correspondam aos tipos dos parâmetros.
- Funções sobrecarregadas são normalmente utilizadas para realizar operações semelhantes que envolvem lógica de programa diferente em diferentes tipos de dados. Se a lógica do programa e as operações forem idênticas para cada tipo de dados, a sobrecarga pode ser realizada mais compacta e convenientemente utilizando templates de função.
- O programador escreve uma única definição de template de função. Dados os tipos de argumentos fornecidos em chamadas para essa função, o C++ gera automaticamente especializações separadas de template de função para tratar cada tipo de chamada apropriadamente. Portanto, definir um único template de função define essencialmente uma família de funções sobrecarregadas.
- Todas as definições de template de função iniciam com a palavra-chave `template` seguida por uma lista de parâmetros de template para o template de função entre colchetes angulares (`< e >`).
- Os parâmetros de tipo formais são marcadores de lugar para tipos fundamentais ou tipos definidos pelo usuário. Esses marcadores de lugar são utilizados para especificar os tipos dos parâmetros da função, especificar o tipo de retorno da função e declarar variáveis dentro do corpo da definição de função.
- Uma função recursiva é uma função que chama a si mesma, direta ou indiretamente.
- Uma função recursiva sabe resolver somente o(s) caso(s) simples(s) ou o(s) chamado(s) caso(s) básico(s). Se a função é chamada com um caso básico, a função simplesmente retorna um resultado.
- Se a função for chamada com um problema mais complexo, em geral, a função divide o problema em duas partes conceituais — uma parte que ela sabe como fazer e uma parte que ela não sabe. Para tornar a recursão realizável, a última parte deve assemelhar-se ao problema original, mas ser uma versão mais simples ou menor dele.
- Para que a recursão, por fim, termine, toda vez que a função chamar a si mesma com uma versão ligeiramente mais simples do problema original, essa sequência de problemas cada vez menor deve, finalmente, convergir para o caso básico.
- A relação de números de Fibonacci sucessivos converge para um valor constante de 1,618.... Esse número ocorre freqüentemente na natureza e foi chamado de relação áurea ou média áurea.
- A iteração e a recursão têm muitas semelhanças: ambas são baseadas em uma instrução de controle, envolvem repetição, envolvem um teste de terminação, aproximam-se gradativamente do término e podem ocorrer infinitamente.
- A recursão tem muitas desvantagens. Ela invoca repetidamente o mecanismo, e conseqüentemente o overhead, das chamadas de função. Isso pode representar um alto custo em tempo de processador como em espaço de memória. Cada chamada recursiva faz com que outra cópia da função (na realidade, somente as variáveis da função) seja criada; isso pode consumir memória considerável.

## Terminologia

|                                      |                                          |                                            |
|--------------------------------------|------------------------------------------|--------------------------------------------|
| & para declarar referência           | declaração de função                     | especificadores de classe de armazenamento |
| abordagem de dividir para conquistar | decoração de nome                        | estouro de pilha                           |
| alias                                | definição de função                      | excluir uma pilha                          |
| argumento-padrão                     | definição de template                    | expressão de tipo misto                    |
| argumentos mais à direita (finais)   | desfiguração de nome                     | fator de escalonamento                     |
| assinatura                           | deslocar um intervalo de números         | fatorial                                   |
| assinatura de função                 | efeito colateral de uma expressão        | Fibonacci, série de                        |
| avaliação recursiva                  | enum, palavra-chave                      | fora de escopo                             |
| bloco externo                        | enumeração                               | função de semente <code>rand</code>        |
| bloco interno                        | escalonamento                            | função de template                         |
| blocos aninhados                     | escopo de arquivo                        | função definida pelo programador           |
| caso(s) básico(s)                    | escopo de bloco                          | função definida pelo usuário               |
| chamada recursiva                    | escopo de classe                         | função global                              |
| classe de armazenamento              | escopo de função                         | função inline                              |
| classe de armazenamento automática   | escopo de namespaces                     | função recursiva                           |
| classe de armazenamento estática     | escopo de protótipo de função            | funções ‘pré-empacotadas’                  |
| coerção de argumento                 | escopo de um identificador               | inicializar uma referência                 |
| colocar em uma pilha                 | especialização de template de função     | <code>inline</code> , palavra-chave        |
| complexidade exponencial             | especificador de classe de armazenamento | inteiros deslocados, escalonados           |
| condição de terminação               | <code>auto</code>                        | invocar uma função                         |
| constante enumerada                  | especificador de classe de armazenamento | iteração                                   |
| convergir para um caso básico        | <code>extern</code>                      | largura de intervalo de número aleatório   |

|                                                   |                                                    |                                                  |
|---------------------------------------------------|----------------------------------------------------|--------------------------------------------------|
| LIFO ( <i>last-in, first-out</i> )                | passar por referência                              | rótulo                                           |
| limites de tamanho de inteiros                    | passar por valor                                   | semente                                          |
| limites de tipo de dados numéricos                | passo de recursão                                  | seqüência de números aleatórios                  |
| linkagem                                          | pilha                                              | sobrecarga de função                             |
| linkagem segura para tipos                        | pilha de chamadas de função                        | sobrecarregando                                  |
| lista de parâmetros de template                   | pilha de execução do programa                      | solução iterativa                                |
| loop infinito                                     | princípio do menor privilégio                      | solução recursiva                                |
| média áurea                                       | procedimento                                       | srand, função                                    |
| métodos                                           | protótipo de função                                | static, especificador de classe de armazenamento |
| modularizar um programa com funções               | protótipos de função obrigatórios                  | static, palavra-chave                            |
| mutable, especificador de classe de armazenamento | quadro de pilha                                    | static, variável local                           |
| nome de função                                    | rand, função                                       | template de função                               |
| nome de função desfigurado                        | RAND_MAX, constante simbólica                      | template, palavra-chave                          |
| nome de uma variável                              | randomizar                                         | terminar chave direita (}) de um bloco           |
| nome do tipo (enumerações)                        | recursão                                           | teste de terminação                              |
| número aleatório                                  | recursão infinita                                  | tipo 'mais alto'                                 |
| números pseudo-aleatórios                         | referência a uma constante                         | 'tipo mais baixo'                                |
| operador unário de resolução de escopo (::)       | referência a uma variável automática               | tipo de uma variável                             |
| otimizar o compilador                             | referência oscilante                               | tipo definido pelo usuário                       |
| overhead de chamada de função                     | register, especificador de classe de armazenamento | truncar parte fracionária de um double           |
| overhead de recursão                              | registro de ativação                               | validar uma chamada de função                    |
| parâmetro                                         | regras de promoção                                 | valor de deslocamento                            |
| parâmetro de referência                           | relação áurea                                      | variável global                                  |
| parâmetro de tipo                                 | repetitividade da função rand                      | variável local automática                        |
| parâmetro de tipo formal                          | retornar uma referência a partir de uma função     | void, tipo de retorno                            |
| parâmetro formal                                  | reutilização de software                           |                                                  |

## Exercícios de revisão

### 6.1 Complete cada uma das sentenças:

- Os componentes de programa em C++ são chamados \_\_\_\_\_ e \_\_\_\_\_.
- Uma função é invocada com um(a) \_\_\_\_\_.
- Uma variável que só é conhecida dentro da função em que ela é definida é chamada de \_\_\_\_\_.
- A instrução \_\_\_\_\_ em uma função chamada passa o valor de uma expressão de volta à função chamadora.
- A palavra-chave \_\_\_\_\_ é utilizada em um cabeçalho de função para indicar que uma função não retorna um valor ou que uma função não contém parâmetros.
- O(A) \_\_\_\_\_ de um identificador é a parte do programa em que o identificador pode ser utilizado.
- As três maneiras de retornar o controle de uma função chamada a um chamador são \_\_\_\_\_, \_\_\_\_\_ e \_\_\_\_\_.
- Um(a) \_\_\_\_\_ permite ao compilador verificar o número, tipos e ordem dos argumentos passados para uma função.
- A função \_\_\_\_\_ é utilizada para produzir números aleatórios.
- A função \_\_\_\_\_ é utilizada para configurar a semente de número aleatório para aleatorizar um programa.
- Os especificadores de classe de armazenamento são mutable, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_ e \_\_\_\_\_.
- Pressupõe-se que as variáveis declaradas em um bloco ou na lista de parâmetros de uma função são da classe de armazenamento \_\_\_\_\_ a menos que especificado de outro modo.
- O especificador de classe de armazenamento \_\_\_\_\_ é uma recomendação para o compilador armazenar uma variável em um dos registros do computador.
- Uma variável declarada fora de qualquer bloco ou função é uma variável \_\_\_\_\_.
- Para uma variável local em uma função reter seu valor entre chamadas para a função, ela deve ser declarada com o especificador de classe de armazenamento \_\_\_\_\_.
- Os seis possíveis escopos de um identificador são \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_ e \_\_\_\_\_.
- Uma função que chama a si própria direta ou indiretamente (isto é, por outra função) é uma função \_\_\_\_\_.
- Uma função recursiva em geral tem dois componentes: um componente que fornece um meio para a recursão terminar testando um caso \_\_\_\_\_ e um que expressa o problema como uma chamada recursiva para um problema ligeiramente mais simples que a chamada original.

- s) Em C++, é possível ter várias funções com o mesmo nome que operam em diferentes tipos ou diferentes números de argumentos. Isso é chamado \_\_\_\_\_ de função.
- t) O(A) \_\_\_\_\_ permite acesso a uma variável global com o mesmo nome de uma variável no escopo atual.
- u) O qualificador \_\_\_\_\_ é utilizado para declarar variáveis de leitura.
- v) Um \_\_\_\_\_ de função permite que uma única função seja definida para realizar uma tarefa em muitos tipos de dados diferentes.

**6.2** Para o programa na Figura 6.40, declare o escopo (escopo de função, de arquivo, de bloco ou de protótipo de função) de cada um dos seguintes elementos:

- a) A variável `x` em `main`.
- b) A variável `y` em `cube`.
- c) A função `cube`.
- d) A função `main`.
- e) O protótipo de função para `cube`.
- f) O identificador `y` no protótipo de função para `cube`.

**6.3** Escreva um programa que testa se os exemplos da chamada de função da biblioteca de matemática mostrados na Figura 6.2 realmente produzem os resultados indicados.

**6.4** Forneça o cabeçalho de função para cada uma das seguintes funções:

- a) A função `hypotenuse`, que aceita dois argumentos de ponto flutuante com dupla precisão `s1` e `s2` e retorna um resultado de ponto flutuante com dupla precisão.
- b) A função `smallest`, que aceita três inteiros, `x`, `y` e `z` e retorna um inteiro.
- c) A função `instructions`, que não recebe argumentos e não retorna um valor. [Nota: Essas funções são comumente utilizadas para exibir instruções ao usuário.]
- d) A função `intToDouble`, que aceita um argumento de inteiro, `number`, e retorna um resultado de ponto flutuante com dupla precisão.

**6.5** Forneça o protótipo de função para cada uma das seguintes:

- a) A função descrita no Exercício 6.4(a).
- b) A função descrita no Exercício 6.4(b).
- c) A função descrita no Exercício 6.4(c).
- d) A função descrita no Exercício 6.4(d).

**6.6** Escreva uma declaração para cada uma das sentenças:

- a) O inteiro `count` que deve ser mantido em um registro. Inicialize `count` como 0.
- b) A variável `lastVal` de ponto flutuante com dupla precisão que deve reter seu valor entre chamadas para a função em que ela é definida.

```

1 // Exercício 6.2: Ex06_02.cpp
2 #include <iostream>
3 using std::cout;
4 using std::endl;
5
6 int cube(int y); // protótipo de função
7
8 int main()
9 {
10 int x;
11
12 for (x = 1; x <= 10; x++) // itera 10 vezes
13 cout << cube(x) << endl; // calcula cubo de x e gera a saída dos resultados
14
15 return 0; // indica terminação bem-sucedida
16 } // fim de main
17
18 // definição de função cube
19 int cube(int y)
20 {
21 return y * y * y;
22 } // fim da função cube

```

**Figura 6.40** Programa para o Exercício 6.2.

**6.7** Localize o erro em cada um dos seguintes segmentos de programa e explique como o erro pode ser corrigido (consulte também o Exercício 6.53):

```
a) int g(void)
{
 cout << "Inside function g" << endl;
 int h(void)
 {
 cout << "Inside function h" << endl;
 }
}

b) int sum(int x, int y)
{
 int result;

 result = x + y;
}

c) int sum(int n)
{
 if (n == 0)
 return 0;
 else
 n + sum(n - 1);
}

d) void f(double a);
{
 float a;
 cout << a << endl;
}

e) void product(void)
{
 int a;
 int b;
 int c;
 int result;
 cout << "Enter three integers: ";
 cin >> a >> b >> c;
 result = a * b * c;
 cout << "Result is " << result;
 return result;
}
```

**6.8** Por que um protótipo de função conteria uma declaração de tipo de parâmetro como `double &?`

**6.9** (Verdadeiro/Falso) Todos os argumentos para as chamadas de função em C++ são passados por valor.

**6.10** Escreva um programa completo que solicita ao usuário o raio de uma esfera e calcula e imprime o volume dessa esfera. Utilize uma função `inline sphereVolume` que retorna o resultado da seguinte expressão:  $(4.0 / 3.0) * 3.14159 * \text{pow}(\text{radius}, 3)$ .

## Respostas dos exercícios de revisão

**6.1** a) funções, classes. b) chamada de função. c) variável local. d) `return`. e) `void`. f) escopo. g) expressão de retorno; `return`; encontrar a chave direita de fechamento de uma função. h) protótipo de função. i) `rand`. j) `srand`. k) `auto`, `register`, `extern`, `static`. l) `auto`. m) `register`. n) global. o) `static`. p) escopo de função, escopo de arquivo, escopo de bloco, escopo de protótipo de função, escopo de classe, escopo de namespaces. q) recursiva. r) básico. s) sobrecarga. t) operador unário de resolução de escopo (`::`). u) `const`. v) `template`.

**6.2** a) escopo de bloco. b) escopo de bloco. c) escopo de arquivo. d) escopo de arquivo. e) escopo de arquivo. f) escopo de protótipo de função.

**6.3** Veja o seguinte programa:



```

1 // Exercício 6.3: Ex06_03.cpp
2 // Testando a biblioteca de funções matemáticas.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::fixed;
7
8 #include <iomanip>
9 using std::setprecision;
10
11 #include <cmath>
12 using namespace std;
13
14 int main()
15 {
16 cout << fixed << setprecision(1);
17
18 cout << "sqrt(" << 900.0 << ") = " << sqrt(900.0)
19 << "\nsqrt(" << 9.0 << ") = " << sqrt(9.0);
20 cout << "\nexp(" << 1.0 << ") = " << setprecision(6)
21 << exp(1.0) << "\nexp(" << setprecision(1) << 2.0
22 << ") = " << setprecision(6) << exp(2.0);
23 cout << "\nlog(" << 2.718282 << ") = " << setprecision(1)
24 << log(2.718282)
25 << "\nlog(" << setprecision(6) << 7.389056 << ") = "
26 << setprecision(1) << log(7.389056);
27 cout << "\nlog10(" << 1.0 << ") = " << log10(1.0)
28 << "\nlog10(" << 10.0 << ") = " << log10(10.0)
29 << "\nlog10(" << 100.0 << ") = " << log10(100.0) ;
30 cout << "\nfabs(" << 13.5 << ") = " << fabs(13.5)
31 << "\nfabs(" << 0.0 << ") = " << fabs(0.0)
32 << "\nfabs(" << -13.5 << ") = " << fabs(-13.5);
33 cout << "\nceil(" << 9.2 << ") = " << ceil(9.2)
34 << "\nceil(" << -9.8 << ") = " << ceil(-9.8);
35 cout << "\nfloor(" << 9.2 << ") = " << floor(9.2)
36 << "\nfloor(" << -9.8 << ") = " << floor(-9.8);
37 cout << "\npow(" << 2.0 << ", " << 7.0 << ") = "
38 << pow(2.0, 7.0) << "\npow(" << 9.0 << ", "
39 << 0.5 << ") = " << pow(9.0, 0.5);
40 cout << setprecision(3) << "\nfmod("
41 << 13.675 << ", " << 2.333 << ") = "
42 << fmod(13.675, 2.333) << setprecision(1);
43 cout << "\nsin(" << 0.0 << ") = " << sin(0.0);
44 cout << "\ncos(" << 0.0 << ") = " << cos(0.0);
45 cout << "\ntan(" << 0.0 << ") = " << tan(0.0) << endl;
46 return 0; // indica terminação bem-sucedida
47 } // fim de main

```

```

sqrt(900.0) = 30.0
sqrt(9.0) = 3.0
exp(1.0) = 2.718282
exp(2.0) = 7.389056
log(2.718282) = 1.0
log(7.389056) = 2.0
log10(1.0) = 0.0
log10(10.0) = 1.0
log10(100.0) = 2.0

```

```

fabs(13.5) = 13.5
fabs(0.0) = 0.0
fabs(-13.5) = 13.5
ceil(9.2) = 10.0
ceil(-9.8) = -9.0
floor(9.2) = 9.0
floor(-9.8) = -10.0
pow(2.0, 7.0) = 128.0
pow(9.0, 0.5) = 3.0
fmod(13.675, 2.333) = 2.010
sin(0.0) = 0.0
cos(0.0) = 1.0
tan(0.0) = 0.0

```

- 6.4** a) `double` hypotenuse( `double` sidel, `double` side2 )  
 b) `int` smallest( `int` x, `int` y, `int` z )  
 c) `void` instructions( `void` ) // em C++ (void) pode ser escrito ()  
 d) `double` intToDouble( `int` number )
- 6.5** a) `double` hypotenuse( `double`, `double` );  
 b) `int` smallest( `int`, `int`, `int` );  
 c) `void` instructions( `void` ); // em C++ (void) pode ser escrito ()  
 d) `double` intToDouble( `int` );
- 6.6** a) `register int` count = 0;  
 b) `static double` lastVal;
- 6.7** a) Erro: A função h é definida na função g.  
 Correção: Mova a definição de h para fora da definição de g.  
 b) Erro: A função supostamente deve retornar um inteiro, mas não o faz.  
 Correção: Exclua a variável result e coloque a seguinte instrução na função:  
`return x + y;`  
 c) Erro: O resultado de `n + sum( n - 1 )` não é retornado; sum retorna um resultado inadequado.  
 Correção: Reescreva a instrução na cláusula else como  
`return n + sum( n - 1 );`  
 d) Erros: O ponto-e-vírgula depois do parêntese direito que inclui a lista de parâmetros, e redefinir o parâmetro a na definição de função.  
 Correções: Exclua o ponto-e-vírgula após o parêntese direito da lista de parâmetros e exclua a declaração `float a;`.  
 e) Erro: A função retorna um valor quando supostamente não deveria.  
 Correção: Elimine a instrução `return`.
- 6.8** Isso cria um parâmetro de referência do tipo 'referência a double' que permite à função modificar a variável original na função chamada.
- 6.9** Falso. O C++ permite passar por referência utilizando parâmetros de referência (e ponteiros, como discutimos no Capítulo 8).
- 6.10** Veja o seguinte programa:

```

1 // Solução do Exercício 6.10: Ex06_10.cpp
2 // Função inline que calcula o volume de uma esfera.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6 using std::endl;
7
8 #include <cmath>
9 using std::pow;
10
11 const double PI = 3.14159; // define a constante global PI
12
13 // calcula o volume de uma esfera

```

```

14 inline double sphereVolume(const double radius)
15 {
16 return 4.0 / 3.0 * PI * pow(radius, 3);
17 } // fim da função inline sphereVolume
18
19 int main()
20 {
21 double radiusValue;
22
23 // solicita o raio ao usuário
24 cout << "Enter the length of the radius of your sphere: ";
25 cin >> radiusValue; // insere o raio
26
27 // utiliza radiusValue para calcular volume da esfera e exibe o resultado
28 cout << "Volume of sphere with radius " << radiusValue
29 << " is " << sphereVolume(radiusValue) << endl;
30 return 0; // indica terminação bem-sucedida
31 } // fim de main

```

## Exercícios

**6.11** Mostre o valor de  $x$  depois que cada uma das seguintes instruções for realizada:

- a)  $x = \text{fabs}(7.5)$
- b)  $x = \text{floor}(7.5)$
- c)  $x = \text{fabs}(0.0)$
- d)  $x = \text{ceil}(0.0)$
- e)  $x = \text{fabs}(-6.4)$
- f)  $x = \text{ceil}(-6.4)$
- g)  $x = \text{ceil}(-\text{fabs}(-8 + \text{floor}(-5.5)))$

**6.12** Um estacionamento cobra uma taxa mínima de \$ 2,00 para estacionar por até três horas. Um adicional de \$ 0,50 por hora não necessariamente inteira é cobrado após as três primeiras horas. A carga máxima para qualquer dado período de 24 horas é \$ 10,00. Suponha que nenhum carro fique estacionado por mais de 24 horas por vez. Escreva um programa que calcula e imprime os custos de estacionamento de cada um dos três clientes que estacionou o carro nessa garagem ontem. Você deve inserir as horas de estacionamento para cada cliente. Seu programa deve imprimir os resultados em um formato tabular elegante e deve calcular e imprimir o total dos recibos de ontem. O programa deve utilizar a função `calculateCharges` para determinar a tarifa para cada cliente. Suas saídas devem aparecer no seguinte formato:

| Car   | Hours | Charge |
|-------|-------|--------|
| 1     | 1.5   | 2.00   |
| 2     | 4.0   | 2.50   |
| 3     | 24.0  | 10.00  |
| TOTAL | 29.5  | 14.50  |

**6.13** Uma aplicação da função `floor` é arredondar um valor para o inteiro mais próximo. A instrução

```
y = floor(x + .5);
```

arredonda o número  $x$  para o inteiro mais próximo e atribui o resultado a  $y$ . Escreva um programa que lê vários números e utiliza a instrução anterior para arredondar cada um desses números para o inteiro mais próximo. Para cada número processado, imprima ambos os números, o original e o arredondado.

**6.14** A função `floor` pode ser utilizada para arredondar um número para uma casa decimal específica. A instrução

```
y = floor(x * 10 + .5) / 10;
```

arredonda  $x$  para a casa decimal (a primeira posição à direita do ponto de fração decimal). A instrução

```
y = floor(x * 100 + .5) / 100;
```

arredonda  $x$  para a casa dos centésimos (isto é, a segunda posição à direita do ponto de fração decimal). Escreva um programa que define quatro funções para arredondar um número  $x$  de várias maneiras:

- a) `roundToInteger( number )`
- b) `roundToTenths( number )`
- c) `roundToHundredths( number )`
- d) `roundToThousandths( number )`

Para cada valor lido, seu programa deve imprimir o valor original, o número arredondado para o inteiro mais próximo, o número arredondado para o décimo mais próximo, o número arredondado para o centésimo mais próximo e o número arredondado para o milésimo mais próximo.

- 6.15** Responda a cada uma das seguintes perguntas:
- a) O que significa escolher números ‘aleatoriamente’?
  - b) Por que a função `rand` é útil para simular jogos de azar?
  - c) Por que você poderia querer aleatorizar um programa utilizando `srand`? Sob quais circunstâncias é desejável não aleatorizar?
  - d) Por que frequentemente é necessário escalonar ou deslocar os valores produzidos por `rand`?
  - e) Por que a simulação computadorizada de situações do mundo real é uma técnica útil?
- 6.16** Escreva instruções que atribuem inteiros aleatórios à variável  $n$  nos seguintes intervalos:
- a)  $1 \leq n \leq 2$
  - b)  $1 \leq n \leq 100$
  - c)  $0 \leq n \leq 9$
  - d)  $1.000 \leq n \leq 1.112$
  - e)  $-1 \leq n \leq 1$
  - f)  $-3 \leq n \leq 11$
- 6.17** Para cada um dos seguintes conjuntos de inteiros, escreva uma única instrução que aleatoriamente imprime um número do conjunto.
- a) 2, 4, 6, 8, 10.
  - b) 3, 5, 7, 9, 11.
  - c) 6, 10, 14, 18, 22.

- 6.18** Escreva uma função `integerPower( base, exponent )` que retorna o valor de

$$base^{exponent}$$

Por exemplo, `integerPower( 3, 4 ) = 3 * 3 * 3 * 3`. Pressuponha que o expoente seja um inteiro não-zero positivo e que a base seja um inteiro. A função `integerPower` deve utilizar `for` ou `while` para controlar o cálculo. Não utilize funções da biblioteca matemática.

- 6.19** (*Hipotenusa*) Defina uma função `hypotenuse` que calcula o comprimento da hipotenusa de um triângulo reto quando os outros dois lados são dados. Utilize essa função em um programa para determinar o comprimento da hipotenusa para cada um dos triângulos mostrados abaixo. A função deve aceitar dois argumentos `double` e retornar a hipotenusa como um `double`.

| Triângulo | Lado 1 | Lado 2 |
|-----------|--------|--------|
| 1         | 3.0    | 4.0    |
| 2         | 5.0    | 12.0   |
| 3         | 8.0    | 15.0   |

- 6.20** Escreva uma função `multiple` que determina para um par de inteiros se o segundo inteiro é um múltiplo do primeiro. A função deve aceitar dois argumentos inteiros e retornar `true` se o segundo for um múltiplo do primeiro e `false`, caso contrário. Utilize essa função em um programa que insere uma série de pares de inteiros.
- 6.21** Escreva um programa que insere uma série de inteiros e os passa um por vez para a função `even`, que utiliza o operador módulo para determinar se um inteiro é par. A função deve aceitar um argumento inteiro e retornar `true` se o inteiro for par e `false`, caso contrário.
- 6.22** Escreva uma função que exibe na margem esquerda da tela um quadrado sólido de asteriscos cujo lado é especificado no parâmetro do tipo inteiro `side`. Por exemplo, se `side` for 4, a função exibirá o seguinte:

```



```

- 6.23** Modifique a função criada no Exercício 6.22 para formar o quadrado a partir de qualquer que seja o caractere contido no parâmetro de caractere `fillCharacter`. Portanto, se `side` for 5 e `fillCharacter` for #, então essa função deve imprimir o seguinte:

```
####
####
####
####
####
```

**6.24** Utilize técnicas semelhantes às aquelas desenvolvidas nos exercícios 6.22 e 6.23 para produzir um programa que representa graficamente uma ampla variedade de formas.

**6.25** Escreva segmentos de programa que realizam cada uma das seguintes instruções:

- Calcule a parte inteira do quociente quando o inteiro *a* é dividido pelo inteiro *b*.
- Calcule o resto inteiro quando o inteiro *a* é dividido pelo inteiro *b*.
- Utilize as partes do programa desenvolvido em (a) e (b) para escrever uma função que insere um inteiro entre 1 e 32767 e o imprime como uma série de dígitos, do qual cada par é separado por dois espaços. Por exemplo, o inteiro 4562 deve ser impresso da seguinte maneira:

```
4 5 6 2
```

**6.26** Escreva uma função que aceita a hora como três argumentos do tipo inteiro (horas, minutos e segundos) e retorna o número de segundos desde a última vez que o relógio ‘deu 12’. Utilize essa função para calcular a quantidade de tempo em segundos entre duas horas, ambas as quais estão dentro de um ciclo de 12 horas.

**6.27** (*Temperaturas Celsius e Fahrenheit*) Implemente as seguintes funções para trabalhar com inteiros:

- A função `celsius` retorna o equivalente em Celsius de uma temperatura em Fahrenheit.
- A função `fahrenheit` retorna o equivalente em Fahrenheit de uma temperatura em Celsius.
- Utilize essas funções para escrever um programa que imprime gráficos para mostrar os equivalentes em Fahrenheit de todas as temperaturas em Celsius de 0 a 100 graus e os equivalentes em Celsius de todas as temperaturas em Fahrenheit de 32 a 212 graus. Imprima as saídas em um formato tabular elegante que minimiza o número de linhas de saída mas permanece legível.

**6.28** Escreva um programa que insere três números de ponto flutuante com dupla precisão e os passa para uma função que retorna o menor número.

**6.29** (*Números perfeitos*) Dizemos que um inteiro é um número perfeito se a soma de seus fatores, incluindo 1 (mas não o próprio número), é igual ao número. Por exemplo, 6 é um número perfeito porque  $6 = 1 + 2 + 3$ . Escreva uma função `perfect` que determina se o parâmetro `number` é um número perfeito. Utilize essa função em um programa que determina e imprime todos os números perfeitos entre 1 e 1.000. Imprima os fatores de cada número perfeito para confirmar se o número é de fato perfeito. Desafie o poder de seu computador testando números muito maiores que 1.000.

**6.30** (*Números primos*) Dizemos que um inteiro é primo se ele é divisível somente por 1 e ele próprio. Por exemplo, 2, 3, 5 e 7 são primos, mas 4, 6, 8 e 9 não o são.

- Escreva uma função que determina se um número é primo.
- Utilize essa função em um programa que determina e imprime todos os números primos entre 2 e 10.000. Quantos desses 10.000 números você realmente tem de testar antes de certificar-se de que encontrou todos os primos?
- Inicialmente você poderia pensar que  $n/2$  é o limite superior que deve testar para ver se um número é primo, mas você precisa ir apenas até a raiz quadrada de *n*. Por quê? Reescreva o programa e execute-o de ambas as maneiras. Estime o melhor desempenho.

**6.31** (*Dígitos invertidos*) Escreva uma função que aceita um valor inteiro e retorna o número com seus dígitos invertidos. Por exemplo, dado o número 7.631, a função deve retornar 1.367.

**6.32** O *máximo divisor comum* (MDC) de dois inteiros é o maior inteiro que é divisível por cada um dos dois números. Escreva uma função `mdc` que retorna o máximo divisor comum de dois inteiros.

**6.33** Escreva uma função `qualityPoints` que insere a média de um aluno e retorna 4 se a média do aluno for 90–100, 3 se a média for 80–89, 2 se a média for 70–79, 1 se a média for 60–69 e 0 se a média for mais baixa que 60.

**6.34** Escreva um programa que simula o lançamento de uma moeda. Para cada lançamento da moeda, o programa deve imprimir `Heads` ou `Tails` (cara ou coroa). Deixe o programa lançar a moeda 100 vezes e conte o número de vezes que cada lado da moeda aparece. Imprima os resultados. O programa deve chamar uma função `flip` separada que não aceita nenhum argumento e retorna 0 para coroa e 1 para cara. [Nota: Se o programa simular realistamente o lançamento de uma moeda, cada lado da moeda deve aparecer aproximadamente metade das vezes.]

**6.35** (*Computadores na educação*) Os computadores estão desempenhando um papel crescente na educação. Escreva um programa que ajuda um aluno da escola primária a aprender multiplicação. Utilize `rand` para produzir dois inteiros a partir de um algarismo positivo. Então ele deve digitar uma pergunta como

Quanto é 6 vezes 7?

Em seguida, o aluno digita a resposta. Seu programa verifica a resposta do aluno. Se estiver correto, imprima "Muito bom!" e então faça outra pergunta de multiplicação. Se a resposta estiver errada, imprima "Não. Tente novamente." e então deixe o aluno tentar a mesma pergunta repetidamente até que, por fim, ele consiga acertar o número.

- 6.36** (*Instrução auxiliada por computador*) O uso de computadores no ensino é referido como instrução auxiliada por computador (*computer-assisted instruction* – CAI). Um problema que se desenvolve em ambientes CAI é a fadiga do aluno. Isso pode ser eliminado variando o diálogo do computador para prender a atenção do aluno. Modifique o programa do Exercício 6.35 de modo que os vários comentários sejam impressos para cada resposta correta e cada resposta incorreta como segue:

Réplicas para uma resposta correta:

Muito bom!  
Excelente!  
Bom trabalho!  
Continue assim!

Réplicas para uma resposta incorreta

Não. Tente novamente.  
Errado. Tente mais uma vez.  
Não desista!  
Não. Continue tentando.

Utilize o gerador de números aleatórios para escolher um número de 1 a 4 a fim de selecionar uma réplica apropriada a cada resposta. Utilize uma instrução `switch` para emitir as respostas.

- 6.37** Sistemas mais sofisticados de instrução auxiliada por computador monitoram o desempenho do aluno durante um período de tempo. A decisão de iniciar um novo tópico é freqüentemente baseada no sucesso do aluno com tópicos anteriores. Modifique o programa do Exercício 6.36 para contar o número de respostas corretas e incorretas digitadas pelo aluno. Depois que o aluno digitar 10 respostas, seu programa deve calcular a porcentagem de respostas corretas. Se a porcentagem for menor que 75%, seu programa deve imprimir "Peça ao seu professor uma ajuda extra" e terminar.

- 6.38** (*Jogo 'Adivinhe o número'*) Escreva um programa que joga 'adivinha o número' como mostrado a seguir: Seu programa escolhe o número a ser adivinhado selecionando um inteiro aleatoriamente no intervalo de 1 a 1.000. O programa então exibe o seguinte:

Tenho um número entre 1 e 1000.  
Você consegue adivinhá-lo?  
Digite sua primeira suposição.

O jogador então digita uma primeira suposição. O programa responde com uma das seguintes frases:

1. Excelente! Você adivinhou o número!  
Quer jogar de novo (s ou n)?  
2. Muito baixo. Tente novamente.  
3. Muito alto. Tente novamente.

Se a suposição do jogador estiver incorreta, o programa deve fazer um loop até o jogador por fim acertar o número. Seu programa deve continuar dizendo ao jogador `Muito alto` ou `Muito baixo` para ajudar o jogador a acertar a resposta.

- 6.39** Modifique o programa do Exercício 6.38 para contar o número de suposições que o jogador faz. Se o número for 10 ou menor, imprima "Você sabe o segredo ou teve sorte!". Se o jogador adivinhar o número em 10 tentativas, então imprima "Ahah! Você sabe o segredo!". Se o jogador fizer mais de 10 suposições, imprima "Você deveria ser capaz de fazer melhor!". Por que não deve haver mais de 10 suposições? Bem, a cada 'boa suposição' o jogador deve ser capaz de eliminar metade dos números. Agora mostre por que qualquer número de 1 a 1.000 pode ser adivinhado em 10 ou menos tentativas.

- 6.40** Escreva uma função recursiva `power( base, exponent )` que, quando invocada, retorna

$base^{exponent}$

Por exemplo, `power( 3, 4 ) = 3 * 3 * 3 * 3`. Suponha que `exponent` é um inteiro maior que ou igual a 1. *Dica:* O passo de recursão utilizaria o relacionamento

$base^{exponent} = base \cdot base^{exponent - 1}$

e a condição de terminação ocorre quando `exponent` é igual a 1 porque

$base^1 = base$

**6.41** (*Série de Fibonacci*) A série de Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

inicia com os termos 0 e 1 e tem a propriedade de que cada termo sucessivo é a soma dos dois termos precedentes. (a) Escreva uma função não recursiva `fibonacci(n)` que calcula o  $n^{\text{ésimo}}$  número de Fibonacci. (b) Determine o maior número de Fibonacci `int` que pode ser impresso no seu sistema. Modifique o programa da parte (a) para utilizar `double` em vez de `int` a fim de calcular e retornar números de Fibonacci e utilize esse programa modificado para repetir a parte (b).

**6.42** (*Torres de Hanói*) Neste capítulo, você estudou funções que podem ser facilmente implementadas tanto recursiva como iterativamente. Neste exercício, apresentamos um problema cuja solução recursiva demonstra a elegância da recursão, e cuja solução iterativa pode não ser tão evidente.

As **Torres de Hanói** são um dos problemas clássicos mais famosos com o qual todo cientista da computação deve lidar. Diz a lenda que, em um templo no Extremo Oriente, os sacerdotes tentavam mover uma pilha de discos de ouro a partir de um pino de diamantes para outro (Figura 6.41). A pilha inicial tem 64 discos sobrepostos em um pino e organizados de baixo para cima por tamanho decrescente. Os sacerdotes tentavam mover a pilha de um pino para outro sob as restrições de que exatamente um disco seria movido por vez e, sob nenhuma circunstância, um disco maior poderia ser colocado em cima de um disco menor. Três pinos eram fornecidos e um deles era utilizado para armazenar discos temporariamente. Supostamente, o mundo acabará quando os sacerdotes completarem sua tarefa, portanto há pouco incentivo para facilitarmos seus esforços.

Vamos assumir que os sacerdotes estão tentando mover os discos do pino 1 para o pino 3. Desejamos desenvolver um algoritmo que imprima a sequência precisa de transferências de discos de um pino para outro.

Se abordássemos esse problema com métodos convencionais, rapidamente ficaríamos desesperados gerenciando os discos. Em vez disso, abordar esse problema com a recursão em mente permite que os passos sejam simples. Mover  $n$  discos pode ser visualizado em termos de mover somente  $n - 1$  discos (e daí a recursão), como segue:

- Mova  $n - 1$  discos do pino 1 para o pino 2, utilizando o pino 3 como área de armazenamento temporário.
- Mova o último disco (o maior) do pino 1 para o pino 3.
- Mova o  $n - 1$  discos do pino 2 para o pino 3, utilizando o pino 1 como área de armazenamento temporário.

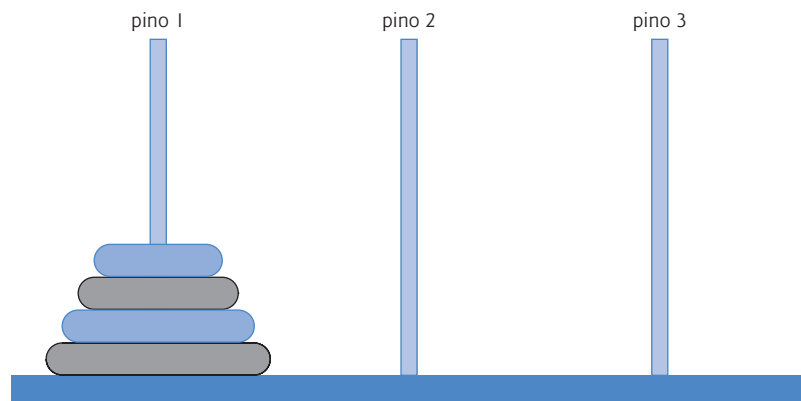
O processo termina quando a última tarefa envolve mover  $n = 1$  disco (isto é, o caso básico). Essa tarefa é realizada simplesmente movendo o disco, sem a necessidade de uma área de armazenamento temporário.

Escreva um programa para resolver o problema Torres de Hanói. Utilize uma função recursiva com quatro parâmetros:

- O número de discos a serem movidos.
- O pino em que esses discos inicialmente estão empilhados.
- O pino para o qual essa pilha de discos deve ser movido.
- O pino a ser utilizado como área de armazenamento temporário.

Seu programa deve imprimir as instruções precisas que ele usará para mover os discos do pino inicial para o pino de destino. Por exemplo, para mover uma pilha de três discos do pino 1 para o pino 3, seu programa deve imprimir a seguinte série de movimentos:

```
1 → 3 (Isso quer dizer mover um disco do pino 1 para o pino 3.)
1 → 2
3 → 2
1 → 3
```



**Figura 6.41** Torres de Hanói para o caso com quatro discos.



2 → 1  
 2 → 3  
 1 → 3

- 6.43** Qualquer programa que pode ser implementado recursivamente pode ser implementado iterativamente, embora às vezes com mais dificuldade e menos clareza. Tente escrever uma versão iterativa das Torres de Hanói. Se você for bem-sucedido, compare sua versão iterativa com a versão recursiva desenvolvida no Exercício 6.42. Investigue questões de desempenho, clareza e sua capacidade de demonstrar a correção dos programas.
- 6.44** (*Visualizando a recursão*) É interessante observar a recursão ‘em ação’. Modifique a função fatorial da Figura 6.29 para imprimir sua variável local e seu parâmetro de chamada recursiva. Para cada chamada recursiva, exiba as saídas em uma linha separada e adicione um nível de recuo. Faça o melhor que você puder para tornar a saída limpa, interessante e significativa. Seu objetivo aqui é projetar e implementar um formato de saída que ajude uma pessoa a entender melhor a recursão. Você pode querer adicionar essas capacidades de exibição aos muitos outros exemplos e exercícios de recursão ao longo de todo este texto.
- 6.45** (*Máximo divisor comum recursivo*) O máximo divisor comum dos inteiros  $x$  e  $y$  é o maior inteiro que é divisível por  $x$  e  $y$ . Escreva uma função recursiva `mdc` que retorna o máximo divisor comum de  $x$  e  $y$ , definido recursivamente como mostrado a seguir: Se  $y$  for igual a 0, então `mdc( x, y )` será  $x$ ; caso contrário, `mdc( x, y )` será `mdc( y, x % y )`, onde `%` é o operador módulo. [Nota: Para esse algoritmo,  $x$  deve ser maior que  $y$ .]
- 6.46** A função `main` pode ser chamada recursivamente em seu sistema? Escreva um programa contendo uma função `main`. Inclua a variável local `static count` e a inicialize como 1. Pós-incrementa e imprima o valor de `count` toda vez que `main` for chamada. Compile seu programa. O que acontece?
- 6.47** Os exercícios 6.35–6.37 desenvolveram um programa de instrução auxiliada por computador para ensinar multiplicação a um aluno da escola primária. Este exercício sugere aprimoramentos nesse programa.
- Modifique o programa para permitir que o usuário insira uma capacidade de nível de graduação. O nível 1 significa utilizar somente números de um único dígito nos problemas, o nível 2 significa utilizar números com dois dígitos etc.
  - Modifique o programa para permitir que o usuário selecione os tipos de problemas aritméticos que ele ou ela deseja estudar. A opção 1 significa problemas de adição somente, 2 significa problemas de subtração somente, 3 significa problemas de multiplicação somente, 4 significa problemas de divisão somente e 5 significa problemas de todos esses tipos misturados aleatoriamente.
- 6.48** Escreva uma função `distance` que calcula a distância entre dois pontos  $(x1, y1)$  e  $(x2, y2)$ . Todos os números e valores de retorno devem ser do tipo `double`.
- 6.49** O que há de errado com o seguinte programa?

```

1 // Exercício 6.49: ex06_49.cpp
2 // O que há de errado com esse programa?
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6
7 int main()
8 {
9 int c;
10
11 if ((c = cin.get()) != EOF)
12 {
13 main();
14 cout << c;
15 } // fim do if
16
17 return 0; // indica terminação bem-sucedida
18 } // fim de main

```

- 6.50** O que o seguinte programa faz?

```

1 // Exercício 6.50: ex06_50.cpp
2 // O que esse programa faz?
3 #include <iostream>
4 using std::cout;

```

```

5 using std::cin;
6 using std::endl;
7
8 int mystery(int, int); // protótipo de função
9
10 int main()
11 {
12 int x, y;
13
14 cout << "Enter two integers: ";
15 cin >> x >> y;
16 cout << "The result is " << mystery(x, y) << endl;
17
18 return 0; // indica terminação bem-sucedida
19 } // fim de main
20
21 // O parâmetro B deve ser um inteiro positivo para evitar recursão infinita
22 int mystery(int a, int b)
23 {
24 if (b == 1) // caso básico
25 return a;
26 else // passo de recursão
27 return a + mystery(a, b - 1);
28 } // fim da função mystery

```

- 6.51** Depois de determinar o que o programa do Exercício 6.50 faz, modifique o programa para funcionar adequadamente depois de remover a restrição de o segundo argumento ser não negativo.
- 6.52** Escreva um programa que testa o maior número de funções da biblioteca de matemática na Figura 6.2 que você puder. Exercite cada uma dessas funções fazendo seu programa imprimir tabelas de valores de retorno para uma diversidade de valores de argumento.
- 6.53** Localize o erro em cada um dos seguintes segmentos de programa e explique como corrigi-los:

a) `float cube( float );` // protótipo de função

```

double cube(float number) // definição de função
{
 return number * number * number;
}

```

b) `register auto int x = 7;`

c) `int randomNumber = srand();`

d) `float y = 123.45678;`  
`int x;`

```

x = y;
cout << static_cast< float >(x) << endl;

```

e) `double square( double number )`

```

{
 double number;
 return number * number;
}

```

f) `int sum( int n )`

```

{
 if (n == 0)
 return 0;
 else
 return n + sum(n);
}

```

- 6.54** Modifique o programa de jogo de dados da Figura 6.11 para permitir apostas. Empacote como uma função a parte do programa que executa o jogo de dados *craps*. Inicialize a variável `bankBalance` como 1.000 dólares. Peça para o jogador inserir um `wager` (aposta). Utilize um loop `while` para verificar se `wager` é menor que ou igual a `bankBalance` e, se não for, solicite ao usuário para inserir `wager` novamente até um `wager` válido ser inserido. Depois que um `wager` correto foi inserido, execute um jogo de dados. Se o jogador ganhar, aumente `bankBalance` por `wager` e imprima o novo `bankBalance`. Se o jogador perder, diminua `bankBalance` por `wager`, imprima o novo `bankBalance`, verifique se `bankBalance` tornou-se zero e, se o for, imprima a mensagem "Sorry. You busted!" [Desculpe. Você perdeu!]. Enquanto o jogo continua, imprima várias mensagens para criar alguns 'diálogos' como "Oh, you're going for broke, huh?" ["Oh, parece que você vai quebrar, hein?"] ou "Aw c'mon, take a chance!" ["Ah, vamos lá, dê uma chance para sua sorte"] ou "You're up big. Now's the time to cash in your chips!" [Você está montado na grana. Agora é hora de trocar essas fichas e embolsar o dinheiro!].
- 6.55** Escreva um programa C++ que solicita ao usuário o raio de um círculo e, então, chama a função `inline circleArea` para calcular a área desse círculo.
- 6.56** Escreva um programa C++ completo com as duas funções alternativas especificadas a seguir, cada uma das quais simplesmente triplica a variável `count` definida em `main`. Então compare e contraste as duas abordagens. Essas duas funções são
- a função `tripleByValue` que passa uma cópia de `count` por valor, triplica a cópia e retorna o novo valor e
  - a função `tripleByReference` que passa `count` por referência via um parâmetro de referência e triplica o valor original de `count` por seu alias (isto é, o parâmetro de referência).
- 6.57** Qual é o propósito do operador unário de resolução de escopo?
- 6.58** Escreva um programa que utiliza um template de função chamado `min` para determinar o menor de dois argumentos. Teste o programa utilizando argumentos do tipo inteiro, caractere e número de ponto flutuante.
- 6.59** Escreva um programa que utiliza um template de função chamado `max` para determinar o maior de três argumentos. Teste o programa utilizando argumentos do tipo inteiro, caractere e número de ponto flutuante.
- 6.60** Determine se os seguintes segmentos de programa contêm erros. Para cada erro, explique como ele pode ser corrigido. [Nota: Para um segmento de programa particular, é possível que nenhum erro esteja presente no segmento.]
- `template < class A >`  
`int sum( int num1, int num2, int num3 )`  
`{`  
`return num1 + num2 + num3;`  
`}`
  - `void printResults( int x, int y )`  
`{`  
`cout << "The sum is " << x + y << '\n';`  
`return x + y;`  
`}`
  - `template < A >`  
`A product( A num1, A num2, A num3 )`  
`{`  
`return num1 * num2 * num3;`  
`}`
  - `double cube( int );`  
`int cube( int );`



*Vai pois agora, escreve isso  
numa tábua perante eles,  
registra-o num livro.*  
Isaías 30:8

*Ir além é tão incerto quanto não  
alcançar o objetivo.*  
Confúcio

*Comece pelo começo, ... e vá  
até ao fim: então, pare.*  
Lewis Carroll

## Arrays e vetores

### OBJETIVOS

Neste capítulo, você aprenderá:

- A utilizar a estrutura de dados de array para representar um conjunto de itens de dados relacionados.
- A utilizar arrays para armazenar, classificar e pesquisar listas e tabelas de valores.
- Como declarar arrays, inicializar arrays e referenciar elementos individuais de arrays.
- Como passar arrays para funções.
- A utilizar técnicas básicas de pesquisa e classificação.
- Como declarar e manipular arrays multidimensionais.
- A utilizar o template `vector` da C++ Standard Library.

- 7.1 Introdução
- 7.2 Arrays
- 7.3 Declarando arrays
- 7.4 Exemplos que utilizam arrays
- 7.5 Passando arrays para funções
- 7.6 Estudo de caso: classe GradeBook utilizando um array para armazenar notas
- 7.7 Pesquisando arrays com pesquisa linear
- 7.8 Classificando arrays por inserção
- 7.9 Arrays multidimensionais
- 7.10 Estudo de caso: classe GradeBook utilizando um array bidimensional
- 7.11 Introdução ao template vector da C++ Standard Library
- 7.12 Estudo de caso de engenharia de software: colaboração entre objetos no sistema ATM (opcional)
- 7.13 Síntese

Resumo | Terminologia | Exercícios de revisão | Respostas dos exercícios de revisão |  
Exercícios | Exercícios com recursão | Exercícios com vector

## 7.1 Introdução

Este capítulo introduz o importante tópico de **estruturas de dados** — coleções de itens de dados relacionados. **Arrays** são estruturas de dados consistindo em itens de dados relacionados do mesmo tipo. Você aprendeu sobre classes no Capítulo 3. No Capítulo 9, discutimos a noção de **estruturas**. Estruturas e classes são capazes de armazenar itens de dados relacionados de tipos possivelmente diferentes. Arrays, estruturas e classes são entidades ‘estáticas’ no sentido de que permanecem com o mesmo tamanho por toda a execução de um programa. (Naturalmente, elas podem ser de uma classe de armazenamento automático e daí serem criadas e destruídas toda vez que o fluxo do programa entra e sai dos blocos em que elas são definidas.)

Depois de discutir como os arrays são declarados, criados e inicializados, este capítulo apresenta uma série de exemplos práticos que demonstra várias manipulações comuns de array. Em seguida, explicamos como strings de caracteres (representadas até agora por objetos `string`) também podem ser representadas por arrays de caracteres. Apresentamos um exemplo de pesquisa de arrays para localizar elementos particulares. O capítulo também introduz uma das aplicações mais importantes da computação — a classificação de dados (isto é, colocar os dados em alguma ordem particular). Duas seções do capítulo aprimoram o estudo de caso da classe `GradeBook` nos capítulos 3–5. Em particular, utilizamos arrays para permitir que a classe mantenha um conjunto de notas na memória e analise as notas de aluno a partir de múltiplos exames em um semestre — duas capacidades que não apareciam nas versões anteriores da classe `GradeBook`. Esses e outros exemplos do capítulo demonstram como os arrays permitem que os programadores organizem e manipulem dados.

O estilo de arrays que utilizamos na maior parte deste capítulo é baseado em ponteiro no estilo do C. (Estudaremos ponteiros no Capítulo 8.) Na seção final deste capítulo e no Capítulo 23, “Standard Template Library (STL)”, abordaremos arrays como objetos completos, com todos os recursos, chamados vetores. Descobriremos que esses arrays baseados em objetos são mais seguros e mais versáteis que os arrays no estilo do C, baseados em ponteiros, que discutimos na primeira parte deste capítulo.

## 7.2 Arrays

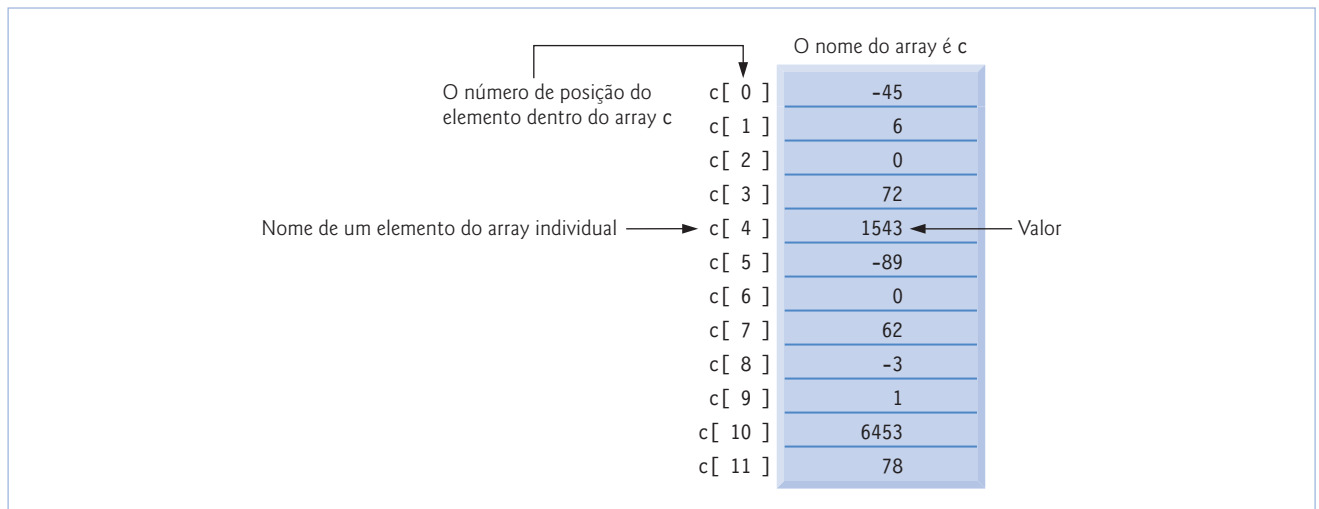
Um array é um grupo consecutivo de posições da memória em que todas elas são do mesmo tipo. Para referir-se a uma particular localização ou elemento no array, especificamos o nome do array e o **número de posição** do elemento particular no array.

A Figura 7.1 mostra um array de inteiros chamado `c`. Esse array contém 12 **elementos**. Um programa referencia qualquer um desses elementos dando o nome do array seguido pelo número da posição do elemento particular entre colchetes (`[]`). O número da posição é chamado mais formalmente de **subscrito** ou **índice** (esse número especifica o número de elementos a partir do início do array). O primeiro elemento em cada array tem **subscrito 0 (zero)** e às vezes é chamado de **zero-ésimo elemento**. Portanto, os elementos do array `c` são `c[ 0 ]` (pronuncia-se ‘c sub zero’), `c[ 1 ]`, `c[ 2 ]` e assim por diante. O subscrito mais alto no array `c` é 11, que é 1 menor que 12 — o número de elementos no array. Os nomes de array seguem as mesmas convenções que outros nomes de variável, isto é, devem ser identificadores.

Um subscrito deve ser um inteiro ou uma expressão do tipo inteiro (que utiliza qualquer tipo inteiro). Se um programa utiliza uma expressão como um subscrito, então o programa avalia a expressão para determinar o subscrito. Por exemplo, se supormos que a variável `a` é igual a 5 e que a variável `b` é igual a 6, então a instrução

```
c[a + b] += 2;
```

adiciona 2 ao elemento do array `c[ 11 ]`. Observe que o nome de um array subscrito é um *lvalue* — ele pode ser utilizado no lado esquerdo de uma atribuição, exatamente como podem os nomes de variáveis que não pertencem a um array.



**Figura 7.1** Array de 12 elementos.

Vamos examinar o array `c` na Figura 7.1 mais atentamente. O **nome** do array de inteiros é `c`. Os 12 elementos do array `c` são referenciados como `c[ 0 ]`, `c[ 1 ]`, `c[ 2 ]`, ..., `c[ 11 ]`. O **valor** de `c[ 0 ]` é -45, o valor de `c[ 1 ]` é 6, o valor de `c[ 2 ]` é 0, o valor de `c[ 7 ]` é 62 e o valor de `c[ 11 ]` é 78. Para imprimir a soma dos valores contidos nos primeiros três elementos do array `c`, escreveríamos

```
cout << c[0] + c[1] + c[2] << endl;
```

Para dividir o valor de `c[ 6 ]` por 2 e atribuir o resultado à variável `x`, escreveríamos

```
x = c[6] / 2;
```



### Erro comum de programação 7.1

É importante notar a diferença entre ‘o sétimo elemento do array’ e ‘elemento 7 do array’. Subscritos de array iniciam em 0, portanto ‘o sétimo elemento do array’ tem um subscrito de 6, enquanto o ‘elemento 7 do array’ tem um subscrito de 7 e na realidade é o oitavo elemento do array. Infelizmente, essa distinção frequentemente é uma fonte de erros off-by-one. Para evitar esses erros, referenciamos elementos do array específicos explicitamente pelo seu nome de array e número de subscrito (por exemplo, `c[ 6 ]` ou `c[ 7 ]`).

Os colchetes utilizados para incluir o subscrito de um array são realmente um operador em C++. Os colchetes têm o mesmo nível de precedência que os parênteses. A Figura 7.2 mostra a precedência e a associatividade dos operadores introduzidos até agora. Observe que foram adicionados colchetes (`[]`) à primeira linha da Figura 7.2. Os operadores são mostrados de cima para baixo na ordem crescente de precedência com sua associatividade e tipo.

## 7.3 Declarando arrays

Os arrays ocupam espaço na memória. O programador especifica o tipo dos elementos e o número de elementos requeridos por um array como segue:

```
tipo nomeDoArray[tamanhoDoArray];
```

e o compilador reserva a quantidade apropriada de memória. O `tamanhoDoArray` deve ser uma constante inteira maior que zero. Por exemplo, para instruir o compilador a reservar 12 elementos para o array de inteiros `c`, utilize a declaração:

```
int c[12]; // c é um array de 12 inteiros
```

A memória pode ser reservada para vários arrays com uma única declaração. A seguinte declaração reserva 100 elementos para o array de inteiros `b` e 27 elementos para o array de inteiros `x`.

```
int b[100], // b é um array de 100 inteiros
 x[27]; // x é um array de 27 inteiros
```



### Boa prática de programação 7.1

Preferimos declarar um array por declaração para legibilidade, modificabilidade e facilidade de comentar.

| Operadores                                           | Associatividade            | Tipo                 |
|------------------------------------------------------|----------------------------|----------------------|
| () []                                                | da esquerda para a direita | mais alto            |
| ++ -- <code>static_cast&lt;tipo&gt;(operando)</code> | da esquerda para a direita | unário (pós-fixado)  |
| ++ -- + - !                                          | da direita para a esquerda | unário (prefixo)     |
| * / %                                                | da esquerda para a direita | multiplicativo       |
| + -                                                  | da esquerda para a direita | aditivo              |
| << >>                                                | da esquerda para a direita | inserção/extração    |
| < <= > >=                                            | da esquerda para a direita | relacional           |
| == !=                                                | da esquerda para a direita | igualdade            |
| &&                                                   | da esquerda para a direita | E lógico             |
|                                                      | da esquerda para a direita | OU lógico            |
| ?:                                                   | da direita para a esquerda | ternário condicional |
| = += -= *= /= %=                                     | da direita para a esquerda | atribuição           |
| ,                                                    | da esquerda para a direita | vírgula              |

**Figura 7.2** Precedência e associatividade de operadores.

Os arrays podem ser declarados para conter valores de qualquer tipo de dados que não referência. Por exemplo, um array do tipo `char` pode ser utilizado para armazenar uma string de caracteres. Até agora, utilizamos objetos `string` para armazenar strings de caracteres. A Seção 7.4 introduz a utilização de arrays de caracteres para armazenar strings. As strings de caracteres e sua semelhança com arrays (um relacionamento do C++ herdado do C), e o relacionamento entre ponteiros e arrays, são discutidos no Capítulo 8.

## 7.4 Exemplos que utilizam arrays

Esta seção apresenta muitos exemplos que demonstram como declarar arrays, como inicializar arrays e como realizar muitas manipulações comuns de arrays.

### *Declarando um array e utilizando um loop para inicializar os elementos do array*

O programa na Figura 7.3 declara array de inteiros `n` de 10 elementos (linha 12). As linhas 15–16 utilizam uma instrução `for` para inicializar os elementos do array como zeros. A primeira instrução de saída (linha 18) exibe os títulos de coluna para as colunas impressas na instrução `for` subsequente (linhas 21–22), que imprime o array em formato tabular. Lembre-se de que `setw` especifica a largura do campo em que somente o próximo valor será enviado para a saída.

### *Inicializando um array em uma declaração com uma lista inicializadora*

Os elementos de um array também podem ser inicializados na declaração do array colocando-se depois do nome do array um sinal de igual e uma lista separada por vírgulas (incluídas entre chaves) de **inicializadores**. O programa na Figura 7.4 utiliza uma **lista inicializadora** para inicializar um array de inteiros com 10 valores (linha 13) e imprime o array em formato tabular (linhas 15–19).

Se houver menos inicializadores que elementos no array, os elementos do array restantes são inicializados como zero. Por exemplo, os elementos do array `n` na Figura 7.3 poderiam ter sido inicializados como zero com a declaração

```
int n[10] = { 0 }; // inicializa elementos do array n como 0
```

A declaração inicializa explicitamente o primeiro elemento zero e inicializa implicitamente os demais nove elementos como zero porque há menos inicializadores que elementos no array. Arrays automáticos não são implicitamente inicializados como zero, embora arrays `static` o sejam. O programador deve pelo menos inicializar o primeiro elemento como zero com uma lista inicializadora para os elementos restantes implicitamente serem configurados como zero. O método de inicialização mostrado na Figura 7.3 pode ser realizado repetidamente enquanto um programa executa.

Se o tamanho do array for omitido de uma declaração com uma lista inicializadora, o compilador determina o número de elementos no array contando o número de elementos na lista inicializadora. Por exemplo,

```
int n[] = { 1, 2, 3, 4, 5 };
```

cria um array de cinco elementos.



```

1 // Figura 7.3: fig07_03.cpp
2 // Inicializando um array.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 int main()
11 {
12 int n[10]; // n é um array de 10 inteiros
13
14 // inicializa elementos do array n como 0
15 for (int i = 0; i < 10; i++)
16 n[i] = 0; // configura elemento na posição i como 0
17
18 cout << "Element" << setw(13) << "Value" << endl;
19
20 // gera saída do valor de cada elemento do array
21 for (int j = 0; j < 10; j++)
22 cout << setw(7) << j << setw(13) << n[j] << endl;
23
24 return 0; // indica terminação bem-sucedida
25 } // fim de main

```

| Element | Value |
|---------|-------|
| 0       | 0     |
| 1       | 0     |
| 2       | 0     |
| 3       | 0     |
| 4       | 0     |
| 5       | 0     |
| 6       | 0     |
| 7       | 0     |
| 8       | 0     |
| 9       | 0     |

**Figura 7.3** Inicializando elementos de um array como zeros e imprimindo o array.

```

1 // Figura 7.4: fig07_04.cpp
2 // Inicializando um array em uma declaração.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 int main()
11 {
12 // utiliza lista inicializadora para inicializar o array n
13 int n[10] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
14

```

**Figura 7.4** Inicializando os elementos de um array em sua declaração.

(continua)

```

15 cout << "Element" << setw(13) << "Value" << endl;
16
17 // gera saída do valor de cada elemento do array
18 for (int i = 0; i < 10; i++)
19 cout << setw(7) << i << setw(13) << n[i] << endl;
20
21 return 0; // indica terminação bem-sucedida
22 } // fim de main

```

| Element | Value |
|---------|-------|
| 0       | 32    |
| 1       | 27    |
| 2       | 64    |
| 3       | 18    |
| 4       | 95    |
| 5       | 14    |
| 6       | 90    |
| 7       | 70    |
| 8       | 60    |
| 9       | 37    |

**Figura 7.4** Inicializando os elementos de um array em sua declaração.

(continuação)

Se o tamanho do array e uma lista inicializadora forem especificados em uma declaração de array, o número de inicializadores deve ser menor que ou igual ao tamanho do array. A declaração de array

```
int n[5] = { 32, 27, 64, 18, 95, 14 };
```

causa um erro de compilação, porque há seis inicializadores e somente cinco elementos do array.



### Erro comum de programação 7.2

*Fornecer mais inicializadores em uma lista inicializadora de array do que o número de elementos existentes no array é um erro de compilação.*



### Erro comum de programação 7.3

*Esquecer de inicializar os elementos de um array cujos elementos deveriam ser inicializados é um erro de lógica.*

*Especificando o tamanho de um array com uma variável constante e configurando elementos do array com cálculos*

A Figura 7.5 configura os elementos de um array de 10 elementos `s` para os inteiros pares 2, 4, 6, ..., 20 (linhas 17–18) e imprime o array em formato tabular (linhas 20–24). Esses números são gerados (linha 18) multiplicando-se cada sucessivo valor do contador de loop por 2 e adicionando 2.

A linha 13 utiliza o **qualificador `const`** para declarar a chamada **variável constante** `arraySize` com o valor 10. Variáveis constantes devem ser inicializadas com uma expressão constante quando são declaradas e não podem ser modificadas depois (como mostrado nas figuras 7.6 e 7.7). Variáveis constantes também são chamadas **constantes identificadas** ou **variáveis de leitura (*read-only*)**.



### Erro comum de programação 7.4

*Não atribuir um valor a uma variável constante quando ela é declarada é um erro de compilação.*



### Erro comum de programação 7.5

*Atribuir um valor a uma variável constante em uma instrução executável é um erro de compilação.*

Variáveis constantes podem ser colocadas em qualquer lugar em que uma expressão constante é esperada. Na Figura 7.5, a variável constante `arraySize` especifica o tamanho do array `s` na linha 15.

```

1 // Figura 7.5: fig07_05.cpp
2 // Configura o array s para os inteiros pares de 2 a 20.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 int main()
11 {
12 // uma variável constante pode ser utilizada para especificar o tamanho do array
13 const int arraySize = 10;
14
15 int s[arraySize]; // array s tem 10 elementos
16
17 for (int i = 0; i < arraySize; i++) // configura os valores
18 s[i] = 2 + 2 * i;
19
20 cout << "Element" << setw(13) << "Value" << endl;
21
22 // gera saída do conteúdo do array s em formato tabular
23 for (int j = 0; j < arraySize; j++)
24 cout << setw(7) << j << setw(13) << s[j] << endl;
25
26 return 0; // indica terminação bem-sucedida
27 } // fim de main

```

| Element | Value |
|---------|-------|
| 0       | 2     |
| 1       | 4     |
| 2       | 6     |
| 3       | 8     |
| 4       | 10    |
| 5       | 12    |
| 6       | 14    |
| 7       | 16    |
| 8       | 18    |
| 9       | 20    |

**Figura 7.5** Gerando valores para serem colocados em elementos de um array.



## Erro comum de programação 7.6

*Somente constantes podem ser utilizadas para declarar o tamanho de arrays automáticos e estáticos. Não utilizar uma constante para esse propósito é um erro de compilação.*

Utilizar variáveis constantes para especificar tamanhos de array torna um programa mais **escalonável**. Na Figura 7.5, a primeira instrução `for` poderia preencher um array de 1000 elementos simplesmente alterando o valor de `arraySize` em sua declaração de 10 para 1000. Se a variável constante `arraySize` não tivesse sido utilizada, teríamos de alterar as linhas 15, 17 e 23 do programa para escalonar o programa a fim de tratar 1000 elementos do array. À medida que os programas crescem, essa técnica torna-se mais útil para escrever programas mais claros e mais fáceis de modificar.



## Observação de engenharia de software 7.1

*Definir o tamanho de cada array como uma variável constante em vez de uma constante literal pode tornar os programas mais escalonáveis.*

```

1 // Figura 7.6: fig07_06.cpp
2 // Utilizando uma variável constante adequadamente inicializada.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9 const int x = 7; // variável constante inicializada
10
11 cout << "The value of constant variable x is: " << x << endl;
12
13 return 0; // indica terminação bem-sucedida
14 } // fim de main

```

The value of constant variable x is: 7

**Figura 7.6** Inicializando e utilizando uma variável constante.

```

1 // Figura 7.7: fig07_07.cpp
2 // Uma variável const deve ser inicializada.
3
4 int main()
5 {
6 const int x; // Erro: x deve ser inicializado
7
8 x = 7; // Erro: não pode modificar a variable const
9
10 return 0; // indica terminação bem-sucedida
11 } // fim de main

```

Mensagem de erro do compilador de linha de comando Borland C++:

```

Error E2304 fig07_07.cpp 6: Constant variable 'x' must be initialized
in function main()
Error E2024 fig07_07.cpp 8: Cannot modify a const object in function main()

```

Mensagem de erro do compilador Microsoft Visual C++.NET:

```

C:\cphttp5_examples\ch07\fig07_07.cpp(6) : error C2734: 'x' : const object
must be initialized if not extern
C:\cphttp5_examples\ch07\fig07_07.cpp(8) : error C2166: l-value specifies
const object

```

Mensagem de erro do compilador GNU C++:

```

fig07_07.cpp:6: error: uninitialized const `x'
fig07_07.cpp:8: error: assignment of read-only variable `x'

```

**Figura 7.7** Variáveis const devem ser inicializadas.



## Boa prática de programação 7.2

Definir o tamanho de um array como uma variável constante em vez de uma constante literal torna os programas mais claros. Essa técnica elimina os chamados *números mágicos*. Por exemplo, mencionar repetidamente o tamanho 10 em código de processamento de array para um array de 10 elementos dá para o número 10 uma importância artificial e infelizmente pode confundir o leitor quando o programa incluir outros números 10 que não têm nada que ver com o tamanho do array.

### Somando os elementos de um array

Freqüentemente, os elementos de um array representam uma série de valores a serem utilizados em um cálculo. Por exemplo, se os elementos de um array representam as notas de exame, um professor pode querer somar os elementos do array e utilizar essa soma para calcular a média da classe. Os exemplos utilizando a classe GradeBook mais adiante no capítulo, nomeadamente as figuras 7.16–7.17 e as figuras 7.23–7.24, utilizam essa técnica.

O programa na Figura 7.8 soma os valores contidos no array de inteiros a de 10 elementos. O programa declara, cria e inicializa o array na linha 10. A instrução for (linhas 14–15) realiza os cálculos. Os valores sendo fornecidos como inicializadores para o array a também poderiam ser lidos para o programa a partir do usuário no teclado ou a partir de um arquivo em disco (consulte o Capítulo 17, “Processamento de arquivo”). Por exemplo, a instrução for

```
for (int j = 0; j < arraySize; j++)
 cin >> a[j];
```

lê um valor por vez do teclado e armazena o valor no elemento a[ j ].

### Utilizando gráficos de barras para exibir dados de array graficamente

Muitos programas apresentam dados graficamente aos usuários. Por exemplo, os valores numéricos são freqüentemente exibidos como barras em um gráfico de barras. Nesse gráfico, as barras mais longas representam os valores numéricos proporcionalmente maiores. Uma maneira simples de exibir os dados numéricos graficamente é utilizar um gráfico de barras que mostra cada valor numérico como uma barra de asteriscos (\*).

Os professores freqüentemente gostam de examinar a distribuição de notas de um exame. Um professor poderia plotar o número de notas em cada uma das várias categorias para visualizar a distribuição das notas. Suponha que as notas fossem 87, 68, 94, 100, 83, 78, 85, 91, 76 e 87. Observe que há uma nota 100, duas notas nos 90s, quatro notas nos 80s, duas notas nos 70s, uma nota nos 60s e nenhuma nota abaixo de 60. Nosso próximo programa (Figura 7.9) armazena esses dados de distribuição de notas em um array de 11 elementos, cada um correspondente a uma categoria de notas. Por exemplo, n[ 0 ] indica o número de notas no intervalo 0–9, n[ 7 ] indica o número de notas no intervalo 70–79 e n[ 10 ] indica o número de notas 100. As duas versões da classe GradeBook mais adiante

```
1 // Figura 7.8: fig07_08.cpp
2 // Calcula a soma dos elementos do array.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9 const int arraySize = 10; // variável constante indicando o tamanho do array
10 int a[arraySize] = { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
11 int total = 0;
12
13 // soma o conteúdo do array a
14 for (int i = 0; i < arraySize; i++)
15 total += a[i];
16
17 cout << "Total of array elements: " << total << endl;
18
19 return 0; // indica terminação bem-sucedida
20 }
```

Total of array elements: 849

**Figura 7.8** Calculando a soma dos elementos de um array.

no capítulo (figuras 7.16–7.17 e figuras 7.23–7.24) contêm código que calcula essas frequências de nota com base em um conjunto de notas. Por enquanto, criamos manualmente o array examinando o conjunto de notas.

O programa lê os números a partir do array e representa as informações graficamente como um gráfico de barras. O programa exibe cada intervalo de notas seguido por uma barra de asteriscos que indica o número de notas nesse intervalo. Para rotular cada barra, as linhas 21-26 geram saída de um intervalo de notas (por exemplo, "70-79: ") com base na variável contadora *i*. A instrução `for`

```

1 // Figura 7.9: fig07_09.cpp
2 // Programa de impressão de gráfico de barras.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 int main()
11 {
12 const int arraySize = 11;
13 int n[arraySize] = { 0, 0, 0, 0, 0, 0, 1, 2, 4, 2, 1 };
14
15 cout << "Grade distribution:" << endl;
16
17 // para cada elemento do array n, gera saída de uma barra do gráfico
18 for (int i = 0; i < arraySize; i++)
19 {
20 // gera a saída do rótulo das barras ("0-9:", ..., "90-99:", "100:")
21 if (i == 0)
22 cout << " 0-9: ";
23 else if (i == 10)
24 cout << " 100: ";
25 else
26 cout << i * 10 << "-" << (i * 10) + 9 << ": ";
27
28 // imprime a barra de asteriscos
29 for (int stars = 0; stars < n[i]; stars++)
30 cout << '*';
31
32 cout << endl; // inicia uma nova linha de saída
33 } // fim do for externo
34
35 return 0; // indica terminação bem-sucedida
36 } // fim de main

```

Grade distribution:

```

0-9:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
100: *

```

**Figura 7.9** Programa de impressão de gráfico de barras.

aninhada (linhas 29–30) gera a saída das barras. Observe a condição de continuação do loop na linha 29 (`stars < n[ i ]`). Toda vez que o programa alcançar o `for` interno, o loop conta de 0 até `n[ i ]`, utilizando assim um valor no array `n` para determinar o número de asteriscos a exibir. Nesse exemplo, `n[ 0 ]–n[ 5 ]` contém zeros porque nenhum aluno recebeu uma nota abaixo de 60. Portanto, o programa não exibe nenhum asterisco perto dos seis primeiros intervalos de notas.



### Erro comum de programação 7.7

*Embora seja possível utilizar a mesma variável de controle em uma instrução `for` e uma segunda instrução `for` aninhada dentro dela, isso é confuso e pode levar a erros de lógica.*

#### Utilizando os elementos de um array como contadores

Às vezes, os programas utilizam as variáveis de contador para resumir dados, como os resultados de uma enquete. Na Figura 6.9, utilizamos os contadores separados em nosso programa de lançamento de dados para monitorar o número de ocorrências de cada face de um dado quando o programa lançou o dado 6.000.000 vezes. Uma versão baseada em array desse programa é mostrada na Figura 7.10.

A Figura 7.10 utiliza o array `frequency` (linha 20) para contar as ocorrências de cada face do dado. A única instrução na linha 26 desse programa substitui a instrução `switch` nas linhas 30–52 da Figura 6.9. A linha 26 utiliza um valor aleatório para determinar qual elemento `frequency` incrementar durante cada iteração do loop. O cálculo na linha 26 produz um subscrito aleatório de 1 a 6, assim o array `frequency` deve ser grande o bastante para armazenar seis contadores. Entretanto, utilizamos um array de sete elementos em que ignoramos `frequency[ 0 ]` — é mais lógico fazer o valor 1 da face do dado incrementar `frequency[ 1 ]` do que `frequency[ 0 ]`. Portanto, o valor de cada face do dado é utilizado como um subscrito para o array `frequency`. Também substituímos as linhas 56–61 da Figura 6.9 fazendo um loop pelo array `frequency` para gerar saída dos resultados (linhas 31–33).

```

1 // Figura 7.10: fig07_10.cpp
2 // Rola um dado de seis lados 6.000.000 vezes.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 #include <cstdlib>
11 using std::rand;
12 using std::srand;
13
14 #include <ctime>
15 using std::time;
16
17 int main()
18 {
19 const int arraySize = 7; // ignora o elemento zero
20 int frequency[arraySize] = { 0 };
21
22 srand(time(0)); // semeia o gerador de número aleatório
23
24 // lança o dado 6.000.000 vezes; usa o valor do dado como índice de frequência
25 for (int roll = 1; roll <= 6000000; roll++)
26 frequency[1 + rand() % 6]++;
27
28 cout << "Face" << setw(13) << "Frequency" << endl;
29
30 // gera a saída do valor de cada elemento do array
31 for (int face = 1; face < arraySize; face++)
32 cout << setw(4) << face << setw(13) << frequency[face]
33 << endl;

```

**Figura 7.10** Programa de rolagem de dados utilizando um array em vez de `switch`.

(continua)



```

34
35 return 0; // indica terminação bem-sucedida
36 } // fim de main

```

| Face | Frequency |
|------|-----------|
| 1    | 1000167   |
| 2    | 1000149   |
| 3    | 1000152   |
| 4    | 998748    |
| 5    | 999626    |
| 6    | 1001158   |

**Figura 7.10** Programa de rolagem de dados utilizando um array em vez de switch.

(continuação)

### Utilizando arrays para resumir resultados de uma enquete

Nosso próximo exemplo (Figura 7.11) utiliza arrays para resumir os resultados de dados coletados em uma enquete. Considere a seguinte declaração do problema:

*Foi pedido a quarenta alunos que avaliassem a qualidade da comida na cantina estudantil em uma escala de 1 a 10 (com 1 significando péssimo e, 10, excelente). Coloque as 40 respostas em um array de inteiros e resuma os resultados da enquete.*

Essa é uma típica aplicação de processamento de array. Queremos resumir o número de respostas de cada tipo (isto é, 1 a 10). O array `responses` (linhas 17–19) é um array de inteiros de 40 elementos das respostas dos alunos à enquete. Observe que o array `responses` é declarado `const`, uma vez que seus valores não mudam (e não devem mudar). Utilizamos um array de 11 elementos `frequency` (linha 22) para contar o número de ocorrências de cada resposta. Cada elemento do array é utilizado como contador para uma das respostas da enquete e inicializado como zero. Como na Figura 7.10, ignoramos `frequency[ 0 ]`.



### Observação de engenharia de software 7.2

*O qualificador `const` deve ser utilizado para impor o princípio do menor privilégio. Utilizar o princípio do menor privilégio adequadamente para projetar software pode reduzir significativamente o tempo de depuração e os efeitos colaterais indevidos e pode tornar um programa mais fácil de modificar e manter.*



### Boa prática de programação 7.3

*Empenhe-se na clareza do programa. Às vezes vale a pena trocar utilização mais eficiente da memória ou tempo de processador em favor de escrever programas mais claros.*



### Dica de desempenho 7.1

*Às vezes considerações de desempenho superam as considerações de clareza.*

A primeira instrução `for` (linhas 26–27) aceita as respostas, uma por vez, do array `responses` e incrementa um dos 10 contadores no array `frequency` (`frequency[ 1 ]` a `frequency[ 10 ]`). A instrução-chave no loop é a linha 27, que incrementa o contador `frequency` adequado, dependendo do valor de `responses[ answer ]`.

Vamos considerar várias iterações do loop `for`. Quando a variável de controle `answer` é 0, o valor de `responses[ answer ]` é o valor de `responses[ 0 ]` (isto é, 1 na linha 17), então o programa interpreta `frequency[ responses[ answer ] ]++` como:

```
frequency[1]++
```

que incrementa o valor no elemento do array 1. Para avaliar a expressão, inicie com o valor no conjunto mais interno de colchetes (`answer`). Uma vez que você sabe o valor de `answer` (que é o valor da variável de controle do loop na linha 26), insira-o na expressão e avalie o próximo conjunto externo de colchetes (isto é, `responses[ answer ]`, que é um valor selecionado do array `responses` nas linhas 17–19). Então, utilize o valor resultante como o subscrito do array `frequency` para especificar qual contador incrementar.

Quando `answer` é 1, `responses[ answer ]` é o valor de `responses[ 1 ]`, que é 2, então o programa interpreta `frequency[ responses[ answer ] ]++` como:

```
frequency[2]++
```

que incrementa o elemento do array 2.

```

1 // Figura 7.11: fig07_11.cpp
2 // Programa de enquete de alunos.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 int main()
11 {
12 // define o tamanho do arrays
13 const int responseSize = 40; // tamanho do array responses
14 const int frequencySize = 11; // tamanho do array frequency
15
16 // coloca as respostas da enquete no array responses
17 const int responses[responseSize] = { 1, 2, 6, 4, 8, 5, 9, 7, 8,
18 10, 1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7,
19 5, 6, 6, 5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };
20
21 // inicializa contadores de frequência como 0
22 int frequency[frequencySize] = { 0 };
23
24 // para cada resposta, seleciona o elemento de respostas e utiliza esse valor
25 // como subscrito de frequência para determinar o elemento a incrementar
26 for (int answer = 0; answer < responseSize; answer++)
27 frequency[responses[answer]]++;
28
29 cout << "Rating" << setw(17) << "Frequency" << endl;
30
31 // gera saída do valor de cada elemento do array
32 for (int rating = 1; rating < frequencySize; rating++)
33 cout << setw(6) << rating << setw(17) << frequency[rating]
34 << endl;
35
36 return 0; // indica terminação bem-sucedida
37 } // fim de main

```

| Rating | Frequency |
|--------|-----------|
| 1      | 2         |
| 2      | 2         |
| 3      | 2         |
| 4      | 2         |
| 5      | 5         |
| 6      | 11        |
| 7      | 5         |
| 8      | 7         |
| 9      | 1         |
| 10     | 3         |

**Figura 7.11** Programa de análise de enquete.

Quando `answer` é 2, `responses[ answer ]` é o valor de `responses[ 2 ]`, que é 6, então o programa interpreta `frequency[ responses[ answer ] ]++` como:

```
frequency[6]++
```

que incrementa o elemento de array 6 e assim por diante. Independentemente do número de respostas processadas na enquete, o programa só exige um array de 11 elementos (ignorando o elemento zero) para resumir o resultado, porque todos os valores de resposta estão entre 1 e 10 e os valores de subscrito de um array de 11 elementos são 0 a 10.

Se os dados no array `responses` contivessem um valor inválido, como 13, o programa teria tentado adicionar 1 a `frequency[ 13 ]`, que está fora dos limites do array. *O C++ não tem nenhuma verificação de limite de array para evitar que o computador referencie um elemento que não existe.* Portanto, um programa que executa pode ‘ultrapassar’ qualquer uma das extremidades de um array sem aviso. O programador deve assegurar que todas as referências de array permaneçam dentro dos limites do array.



### Erro comum de programação 7.8

*Referenciar um elemento fora dos limites de array é um erro de lógica de tempo de execução. Não é um erro de sintaxe.*



### Dica de prevenção de erro 7.1

*Ao fazer um loop por um array, o subscrito de array nunca deveria ir abaixo de 0 e sempre deveria ser menor que o número total de elementos no array (um menor que o tamanho do array). Certifique-se de que a condição de terminação do loop impede o acesso a elementos fora desse intervalo.*



### Dica de portabilidade 7.1

*Os efeitos (normalmente sérios) de referenciar elementos fora dos limites do array são dependentes do sistema. Frequentemente isso resulta em alterações no valor de uma variável não relacionada ou em um erro fatal que termina a execução do programa.*

O C++ é uma linguagem extensível. A Seção 7.11 apresenta o template `vector` do C++ Standard Library, que permite que os programadores realizem muitas operações que não estão disponíveis nos arrays predefinidos do C++. Por exemplo, seremos capazes de comparar `vectors` diretamente e atribuir um `vector` a outro. No Capítulo 11, estendemos o C++ mais ainda implementando um array como uma classe definida pelo próprio usuário. Essa nova definição de array nos permitirá inserir e gerar saída de arrays inteiros com `cin` e `cout`, inicializar arrays quando eles são criados, evitar acesso a elementos do array fora do intervalo e alterar o intervalo de subscritos (e até seu tipo de subscrito) de modo que o primeiro elemento de um array não seja obrigado a ser o elemento 0. Seremos capazes até de utilizar subscritos não inteiros.



### Dica de prevenção de erro 7.2

*No Capítulo 11, veremos como desenvolver uma classe que representa um ‘array inteligente’, que verifica que todas as referências a subscritos estejam dentro dos limites em tempo de execução. Utilizar tipos de dados assim inteligentes ajuda a eliminar bugs.*

### Utilizando arrays de caracteres para armazenar e manipular strings

Até este ponto, discutimos somente arrays de inteiros. Entretanto, os arrays podem ser de qualquer tipo. Agora introduziremos o armazenamento de strings de caracteres em arrays de caracteres. Lembre-se de que, desde o Capítulo 3, utilizamos objetos `string` para armazenar strings de caracteres, como o nome do curso em nossa classe `GradeBook`. Uma string como ‘hello’ é na realidade um array de caracteres. Embora objetos `string` sejam convenientes para utilizar e reduzir o potencial para erros, arrays de caracteres que representam strings têm vários recursos únicos, que discutimos nesta seção. À medida que continua seu estudo de C++, você pode encontrar capacidades do C++ que exigem utilizar arrays de caracteres de preferência a objetos `string`. Você também pode ser solicitado a atualizar código existente utilizando arrays de caracteres.

Um array de caracteres pode ser inicializado utilizando um literal string. Por exemplo, a declaração

```
char string1[] = "first";
```

inicializa os elementos do array `string1` para os caracteres individuais no literal string “first”. O tamanho do array `string1` na declaração precedente é determinado pelo compilador com base no comprimento da string. É importante observar que a string “first” contém cinco caracteres *mais* um caractere especial de terminação de string chamado **caractere nulo**. Portanto, o array `string1` na realidade contém seis elementos. A representação de constante de caractere do caractere nulo é ‘\0’ (barras invertidas seguidas por zero). Todas as strings representadas por arrays de caracteres acabam com esse caractere. Um array de caracteres representando uma string sempre deve ser declarado com um tamanho grande o suficiente para armazenar o número de caracteres na string e o caractere nulo de terminação.

Os arrays de caracteres também podem ser inicializados com constantes de caractere individuais em uma lista inicializadora. A declaração precedente é equivalente à forma mais tediosa:

```
char string1[] = { 'f', 'i', 'r', 's', 't', '\0' };
```

Observe o uso de aspas únicas para delinear cada caractere constante. Além disso, observe que fornecemos explicitamente o caractere nulo de terminação como o último valor inicializador. Sem ele, esse array simplesmente representaria um array de caracteres, não uma string. Como discutimos no Capítulo 8, não fornecer um caractere nulo de terminação a uma string pode causar erros de lógica.

Como uma string é um array de caracteres, podemos acessar caracteres individuais em uma string diretamente com a notação de subscrito de array. Por exemplo, `string1[ 0 ]` é o caractere 'f', `string1[ 3 ]` é o caractere 's', e `string1[ 5 ]` é o caractere nulo.

Também podemos inserir uma string diretamente em um array de caracteres a partir do teclado utilizando `cin` e `>>`. Por exemplo, a declaração

```
char string2[20];
```

cria um array de caracteres capaz de armazenar uma string de 19 caracteres e um caractere nulo de terminação. A instrução

```
cin >> string2;
```

lê uma string inserida pelo teclado para `string2` e acrescenta o caractere nulo ao fim da entrada de string feita pelo usuário. Observe que a instrução precedente fornece somente o nome do array e nenhuma informação sobre o tamanho do array. É de responsabilidade do programador assegurar que o array para o qual a string é lida é capaz de armazenar qualquer string que o usuário digite no teclado. Por padrão, `cin` lê caracteres do teclado até que o primeiro caractere de espaço em branco seja encontrado — independentemente do tamanho do array. Portanto, inserir dados com `cin` e `>>` pode inserir dados além do fim do array (consulte a Seção 8.13 para informações sobre como evitar inserção além do fim de um array `char`).



### Erro comum de programação 7.9

*Não fornecer a `cin >>` um array de caracteres suficientemente grande para armazenar uma string digitada no teclado pode resultar em perda de dados em um programa e outros erros sérios de tempo de execução.*

Um array de caracteres representando uma string terminada por caractere nulo pode ser enviado para a saída com `cout` e `<<`. A instrução

```
cout << string2;
```

imprime o array `string2`. Observe que `cout <<`, assim como `cin >>`, não se importa com o tamanho do array de caracteres. Os caracteres da string são enviados para a saída até que um caractere nulo de terminação seja encontrado. [Nota: `cin` e `cout` supõem que os arrays de caracteres devem ser processados como strings terminadas por caracteres nulos; `cin` e `cout` não fornecem capacidades semelhantes de processamento de entrada e saída para outros tipos de array.]

A Figura 7.12 demonstra como inicializar um array de caracteres com um literal string, ler uma string para um array de caracteres, imprimir um array de caracteres como uma string e acessar caracteres individuais de uma string.

As linhas 23–24 da Figura 7.12 utilizam uma instrução `for` para iterar pelo `string1` o array e imprime os caracteres individuais separados por espaços. A condição na instrução `for`, `string1[ i ] != '\0'`, é verdadeira até que o loop encontra o caractere nulo de terminação da string.

### Arrays locais estáticos e arrays locais automáticos

O Capítulo 6 discutiu o especificador de classe de armazenamento `static`. Uma variável local `static` em uma definição de função existe até o fim do programa, mas é visível somente no corpo da função.



### Dica de desempenho 7.2

*Podemos aplicar `static` para uma declaração local de array de modo que o array não seja criado e inicializado toda vez que o programa chama a função e não seja destruído toda vez que a função termina no programa. Isso pode melhorar o desempenho, especialmente quando utilizando arrays grandes.*

Um programa inicializa arrays locais `static` quando suas declarações são encontradas pela primeira vez. Se um array `static` não é inicializado explicitamente pelo programador, cada elemento desse array é inicializado como zero pelo compilador quando o array é criado. Lembre-se de que o C++ não realiza essa inicialização-padrão para variáveis automáticas.

A Figura 7.13 demonstra a função `staticArrayInit` (linhas 25–41) com um array local `static` (linha 28) e a função `automaticArrayInit` (linhas 44–60) com um array local automático (linha 47).

A função `staticArrayInit` é chamada duas vezes (linhas 13 e 17). O array local `static` é inicializado como zero pelo compilador na primeira vez que a função é chamada. A função imprime o array, adiciona 5 a cada elemento e imprime o array novamente. Na segunda vez que a função é chamada, o array `static` contém os valores modificados armazenados durante a primeira chamada da função. A função `automaticArrayInit` também é chamada duas vezes (linhas 14 e 18). Os elementos do array local automático são inicializados (linha 47) com os valores 1, 2 e 3. A função imprime o array, adiciona 5 a cada elemento e imprime o array novamente. Na segunda vez que a função é chamada, os elementos do array são reinicializados como 1, 2 e 3. O array tem classe de armazenamento automática, então o array é recriado durante cada chamada a `automaticArrayInit`.



### Erro comum de programação 7.10

*Supor que elementos de um array `static` local de uma função são inicializados cada vez que a função é chamada pode levar a erros de lógica em um programa.*

```

1 // Figura 7.12: fig07_12.cpp
2 // Tratando arrays de caracteres como strings.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 int main()
9 {
10 char string1[20]; // reserva 20 caracteres
11 char string2[] = "string literal"; // reserva 15 caracteres
12
13 // lê a string fornecida pelo usuário para o array string1
14 cout << "Enter the string \"hello there\": ";
15 cin >> string1; // lê "hello" [o espaço termina a entrada]
16
17 // gera a saída de strings
18 cout << "string1 is: " << string1 << "\nstring2 is: " << string2;
19
20 cout << "\nstring1 with spaces between characters is:\n";
21
22 // caracteres de saída até que caractere nulo é alcançado
23 for (int i = 0; string1[i] != '\0'; i++)
24 cout << string1[i] << ' ';
25
26 cin >> string1; // lê "there"
27 cout << "\nstring1 is: " << string1 << endl;
28
29 return 0; // indica terminação bem-sucedida
30 } // fim de main

```

```

Enter the string "hello there": hello there
string1 is: hello
string2 is: string literal
string1 with spaces between characters is:
h e l l o
string1 is: there

```

**Figura 7.12** Os arrays de caracteres processados como strings.

```

1 // Figura 7.13: fig07_13.cpp
2 // Arrays estáticos são inicializados como zero.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 void staticArrayInit(void); // protótipo de função
8 void automaticArrayInit(void); // protótipo de função
9
10 int main()
11 {
12 cout << "First call to each function:\n";
13 staticArrayInit();
14 automaticArrayInit();

```

**Figura 7.13** Inicialização de array static e inicialização de array automático.

(continua)

```

15
16 cout << "\n\nSecond call to each function:\n";
17 staticArrayInit();
18 automaticArrayInit();
19 cout << endl;
20
21 return 0; // indica terminação bem-sucedida
22 } // fim de main
23
24 // função para demonstrar um array local estático
25 void staticArrayInit(void)
26 {
27 // inicializa elementos como 0 na primeira vez que a função é chamada
28 static int array1[3]; // array local estático
29
30 cout << "\nValues on entering staticArrayInit:\n";
31
32 // gera saída do conteúdo de array1
33 for (int i = 0; i < 3; i++)
34 cout << "array1[" << i << "] = " << array1[i] << " ";
35
36 cout << "\nValues on exiting staticArrayInit:\n";
37
38 // modifica e gera saída do conteúdo de array1
39 for (int j = 0; j < 3; j++)
40 cout << "array1[" << j << "] = " << (array1[j] += 5) << " ";
41 } // fim da função staticArrayInit
42
43 // função para demonstrar um array local automático
44 void automaticArrayInit(void)
45 {
46 // inicializa elementos toda vez que a função é chamada
47 int array2[3] = { 1, 2, 3 }; // array local automático
48
49 cout << "\n\nValues on entering automaticArrayInit:\n";
50
51 // gera saída do conteúdo de array2
52 for (int i = 0; i < 3; i++)
53 cout << "array2[" << i << "] = " << array2[i] << " ";
54
55 cout << "\nValues on exiting automaticArrayInit:\n";
56
57 // modifica e gera saída do conteúdo de array2
58 for (int j = 0; j < 3; j++)
59 cout << "array2[" << j << "] = " << (array2[j] += 5) << " ";
60 } // fim da função automaticArrayInit

```

First call to each function:

Values on entering staticArrayInit:  
array1[0] = 0 array1[1] = 0 array1[2] = 0  
Values on exiting staticArrayInit:  
array1[0] = 5 array1[1] = 5 array1[2] = 5

Values on entering automaticArrayInit:  
array2[0] = 1 array2[1] = 2 array2[2] = 3

**Figura 7.13** Inicialização de array static e inicialização de array automático.

(continua)

```

Values on exiting automaticArrayInit:
array2[0] = 6 array2[1] = 7 array2[2] = 8

Second call to each function:

Values on entering staticArrayInit:
array1[0] = 5 array1[1] = 5 array1[2] = 5
Values on exiting staticArrayInit:
array1[0] = 10 array1[1] = 10 array1[2] = 10

Values on entering automaticArrayInit:
array2[0] = 1 array2[1] = 2 array2[2] = 3
Values on exiting automaticArrayInit:
array2[0] = 6 array2[1] = 7 array2[2] = 8

```

**Figura 7.13** Inicialização de array static e inicialização de array automático.

(continuação)

## 7.5 Passando arrays para funções

Para passar um argumento array a uma função, especifique o nome do array sem colchetes. Por exemplo, se o array `hourlyTemperatures` foi declarado como

```
int hourlyTemperatures[24];
```

a chamada de função

```
modifyArray(hourlyTemperatures, 24);
```

passa o array de `hourlyTemperatures` e seu tamanho à função `modifyArray`. Ao passar um array a uma função, o tamanho do array normalmente é passado também, assim a função pode processar o número específico de elementos no array. (Caso contrário, precisaríamos embutir esse conhecimento na própria função chamada ou, pior ainda, colocar o tamanho do array em uma variável global.) Na Seção 7.11, quando apresentamos o template `vector` da C++ Standard Library para representar um tipo de array mais robusto, você verá que o tamanho de um `vector` é predefinido — cada objeto `vector` ‘conhece’ seu próprio tamanho, que pode ser obtido invocando-se a função-membro `size` do objeto `vector`. Portanto, quando passarmos um objeto `vector` a uma função, não teremos de passar o tamanho do `vector` como um argumento.

O C++ passa arrays a funções por referência — as funções chamadas podem modificar os valores dos elementos nos arrays originais dos chamadores. O valor do nome do array é o endereço na memória do computador do primeiro elemento do array. Como o endereço inicial do array é passado, a função chamada sabe precisamente onde o array está armazenado na memória. Portanto, quando a função chamada modifica elementos do array no seu corpo, ela está modificando os elementos reais do array em suas posições originais na memória.



### Dica de desempenho 7.3

*Passar arrays por referência faz sentido por razões de desempenho. Se arrays fossem passados por valor, uma cópia de cada elemento seria passada. Para grandes arrays frequentemente passados, isso seria demorado e exigiria armazenamento considerável para as cópias dos elementos do array.*



### Observação de engenharia de software 7.3

*É possível passar um array por valor (utilizando um truque simples que explicamos no Capítulo 22) — mas isso raramente é feito.*

Embora arrays inteiros sejam passados por referência, elementos do array individuais são passados por valor exatamente como variáveis simples o são. Esses fragmentos de dados simples são chamados **escalares** ou **quantidades escalares**. Para passar um elemento de um array para uma função, utilize o nome subscrito do elemento do array como um argumento na chamada de função. No Capítulo 6, mostramos como passar escalares (isto é, variáveis individuais e elementos do array) por referência com referências. No Capítulo 8, mostramos como passar escalares por referência com ponteiros.

Para uma função receber um array por meio de uma chamada de função, a lista de parâmetros da função deve especificar que a função espera receber um array. Por exemplo, o cabeçalho da função `modifyArray` poderia ser escrito como:

```
void modifyArray(int b[], int arraySize)
```

indicando que `modifyArray` espera receber o endereço de um array de inteiros no parâmetro `b` e o número de elementos do array no parâmetro `arraySize`. O tamanho do array não é requerido entre os colchetes do array. Se for incluído, o compilador irá ignorá-lo.



Como o C++ passa arrays a funções por referência, quando a função chamada utiliza o nome de array `b`, ela estará de fato referenciando o array real no chamador (isto é, o array `hourlyTemperatures` discutido no começo desta seção).

Observe a aparência estranha do protótipo de função para `modifyArray`

```
void modifyArray(int [], int);
```

Esse protótipo poderia ter sido escrito assim:

```
void modifyArray(int anyArrayName[], int anyVariableName);
```

mas, como aprendemos no Capítulo 3, compiladores C++ ignoram nomes de variáveis em protótipos. Lembre-se, o protótipo informa ao compilador o número de argumentos e o tipo de cada argumento (na ordem em que se espera que os argumentos apareçam).

O programa na Figura 7.14 demonstra a diferença entre passar um array inteiro e passar um elemento do array. As linhas 22–23 imprimem os cinco elementos originais do array de inteiros `a`. A linha 28 passa `a` e seu tamanho à função `modifyArray` (linhas 45–50), que multiplica cada um dos elementos de `a` por 2 (por meio do parâmetro `b`). Depois, as linhas 32–33 imprimem o array `a` novamente em `main`. Enquanto mostra a saída, os elementos de `a` são de fato modificados por `modifyArray`. Em seguida, a linha 36 imprime o valor do escalar `a[ 3 ]`, então a linha 38 passa o elemento `a[ 3 ]` à função `modifyElement` (linhas 54–58), que multiplica seu parâmetro por 2 e imprime o novo valor. Observe que, quando a linha 39 novamente imprime `a[ 3 ]` em `main`, o valor não foi modificado, porque os elementos do array individuais são passados por valor.

Há situações em seus programas nas quais uma função não deve ter permissão para modificar elementos do array. O C++ fornece o qualificador de tipo `const` que pode ser utilizado para evitar modificação de valores do array no chamador por meio de código em uma função chamada. Quando uma função especifica um parâmetro de array que é precedido pelo qualificador `const`, os elementos do array tornam-se constantes no corpo da função, e qualquer tentativa de modificar um elemento do array no corpo da função resulta em um erro de compilação. Isso permite ao programador evitar modificação acidental de elementos do array no corpo da função.

```
1 // Figura 7.14: fig07_14.cpp
2 // Passando arrays e elementos de array individuais a funções.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 void modifyArray(int [], int); // parece estranho
11 void modifyElement(int);
12
13 int main()
14 {
15 const int arraySize = 5; // tamanho do array a
16 int a[arraySize] = { 0, 1, 2, 3, 4 }; // inicializa o array a
17
18 cout << "Effects of passing entire array by reference:"
19 << "\n\nThe values of the original array are:\n";
20
21 // gera saída de elementos do array original
22 for (int i = 0; i < arraySize; i++)
23 cout << setw(3) << a[i];
24
25 cout << endl;
26
27 // passa o array a para modifyArray por referência
28 modifyArray(a, arraySize);
29 cout << "The values of the modified array are:\n";
30
31 // gera saída de elementos do array modificado
32 for (int j = 0; j < arraySize; j++)
33 cout << setw(3) << a[j];
```

**Figura 7.14** Passando arrays e elementos de array individuais a funções.

(continua)

```

34
35 cout << "\n\nEffects of passing array element by value:"
36 << "\n\na[3] before modifyElement: " << a[3] << endl;
37
38 modifyElement(a[3]); // passa elemento do array a[3] por valor
39 cout << "a[3] after modifyElement: " << a[3] << endl;
40
41 return 0; // indica terminação bem-sucedida
42 } // fim de main
43
44 // na função modifyArray, "b" aponta para o array original "a" na memória
45 void modifyArray(int b[], int sizeofArray)
46 {
47 // multiplica cada elemento do array por 2
48 for (int k = 0; k < sizeofArray; k++)
49 b[k] *= 2;
50 } // fim da função modifyArray
51
52 // na função modifyElement, "e" é uma cópia local do
53 // elemento do array a[3] passado de main
54 void modifyElement(int e)
55 {
56 // multiplica parâmetro por 2
57 cout << "Value of element in modifyElement: " << (e *= 2) << endl;
58 } // fim da função modifyElement

```

Effects of passing entire array by reference:

The values of the original array are:

0 1 2 3 4

The values of the modified array are:

0 2 4 6 8

Effects of passing array element by value:

a[3] before modifyElement: 6

Value of element in modifyElement: 12

a[3] after modifyElement: 6

**Figura 7.14** Passando arrays e elementos de array individuais a funções.

(continuação)

A Figura 7.15 demonstra o qualificador `const`. A função `tryToModifyArray` (linhas 21–26) é definida com parâmetro `const int b[]`, que especifica que o array `b` é constante e não pode ser modificado. Cada uma das três tentativas por parte da função de modificar os elementos do array `b` (linhas 23–25) resultam em um erro de compilação. O compilador Microsoft Visual C++ .NET, por exemplo, produz o erro `'l-value specifies const object.'` [Nota: O padrão C++ define um ‘objeto’ como qualquer ‘região de armazenamento’, incluindo assim variáveis ou elementos de array dos tipos de dados fundamentais, bem como instâncias de classes (o que temos chamado de objetos).] Essa mensagem indica que utilizar um objeto `const` (por exemplo, `b[ 0 ]`) como um *lvalue* é um erro — você não pode atribuir um novo valor a um objeto `const` colocando-o à esquerda de um operador de atribuição. Observe que as mensagens de erro de compilador variam entre compiladores (como mostrado na Figura 7.15). O qualificador `const` será discutido novamente no Capítulo 10.



### Erro comum de programação 7.11

*Esquecer que arrays no chamador são passados por referência, e daí poderem ser modificados em funções chamadas, pode resultar em erros de lógica.*



## Observação de engenharia de software 7.4

Aplicar o qualificador de tipo `const` a um parâmetro de array em uma definição de função para impedir que o array original seja modificado no corpo da função é outro exemplo do princípio de menor privilégio. As funções não devem receber a capacidade de modificar um array a menos que seja absolutamente necessário.

```

1 // Figura 7.15: fig07_15.cpp
2 // Demonstrando o qualificador de tipo const.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 void tryToModifyArray(const int []); // protótipo de função
8
9 int main()
10 {
11 int a[] = { 10, 20, 30 };
12
13 tryToModifyArray(a);
14 cout << a[0] << ' ' << a[1] << ' ' << a[2] << '\n';
15
16 return 0; // indica terminação bem-sucedida
17 } // fim de main
18
19 // Na função tryToModifyArray, "b" não pode ser utilizado
20 // para modificar o array original "a" em main.
21 void tryToModifyArray(const int b[])
22 {
23 b[0] /= 2; // error
24 b[1] /= 2; // error
25 b[2] /= 2; // error
26 } // fim da função tryToModifyArray

```

Mensagem de erro do compilador de linha de comando Borland C++:

```

Error E2024 fig07_15.cpp 23: Cannot modify a const object
 in function tryToModifyArray(const int * const)
Error E2024 fig07_15.cpp 24: Cannot modify a const object
 in function tryToModifyArray(const int * const)
Error E2024 fig07_15.cpp 25: Cannot modify a const object
 in function tryToModifyArray(const int * const)

```

Mensagem de erro do compilador Microsoft Visual C++.NET:

```

C:\cphttp5_examples\ch07\fig07_15.cpp(23) : error C2166: l-value specifies
 const object
C:\cphttp5_examples\ch07\fig07_15.cpp(24) : error C2166: l-value specifies
 const object
C:\cphttp5_examples\ch07\fig07_15.cpp(25) : error C2166: l-value specifies
 const object

```

Mensagem de erro do compilador GNU C++:

```

fig07_15.cpp:23: error: assignment of read-only location
fig07_15.cpp:24: error: assignment of read-only location
fig07_15.cpp:25: error: assignment of read-only location

```

**Figura 7.15** Qualificador de tipo `const` aplicado a um parâmetro de array.

## 7.6 Estudo de caso: classe GradeBook utilizando um array para armazenar notas

Esta seção desenvolve ainda mais a classe GradeBook, introduzida no Capítulo 3 e expandida nos capítulos 4–6. Lembre-se de que essa classe representa um livro de notas utilizado por um professor para armazenar e analisar um conjunto de notas de alunos. Versões anteriores da classe processavam um conjunto de notas inseridas pelo usuário, mas não mantinham os valores das notas individuais em membros de dados da classe. Portanto, os cálculos de repetição exigem que o usuário insira as mesmas notas novamente. Uma maneira de resolver esse problema seria armazenar cada nota inserida em um membro de dados da classe. Por exemplo, poderíamos criar os membros de dados grade1, grade2, ..., grade10 na classe GradeBook para armazenar 10 notas de alunos. Entretanto, o código para somar as notas e determinar a média de classe seria complicado. Nesta seção, resolvemos esse problema armazenando as notas em um array.

### Armazenando notas de aluno em um array na classe *GradeBook*

A versão da classe GradeBook (figuras 7.16–7.17) apresentada aqui utiliza um array de inteiros para armazenar as notas de vários alunos em um único exame. Isso elimina a necessidade de inserir o mesmo conjunto de notas repetidamente. O array grades é declarado como um membro de dados na linha 29 da Figura 7.16 — portanto, cada objeto GradeBook mantém seu próprio conjunto de notas.

Observe que o tamanho do array na linha 29 da Figura 7.16 é especificado pelo membro de dados public const static students (declarado na linha 13). Esse membro de dados é public e, portanto, acessível aos clientes da classe. Logo veremos um exemplo de um programa-cliente que utiliza essa constante. Declarar students com o qualificador const indica que esse membro de dados é constante — seu valor não pode ser alterado depois de ser inicializado. A palavra-chave static nessa declaração de variável indica que o membro de dados é compartilhado por todos os objetos da classe — todos os objetos GradeBook armazenam notas para o mesmo número de alunos. Lembre-se, a partir da Seção 3.6, de que, quando cada objeto de uma classe mantém sua própria cópia de um atributo, a variável que representa o atributo também é conhecida como membro de dados — cada objeto (instância) da classe tem uma cópia separada da variável na memória. Há variáveis para as quais cada objeto de uma classe não tem uma cópia separada. Esse é o caso com membros de dados static, que também são conhecidos como **variáveis de classe**. Quando objetos de uma classe contendo membros de dados static são criados, todos os objetos dessa classe compartilham uma cópia dos membros de dados static da classe.

```
1 // Figura 7.16: GradeBook.h
2 // Definição da classe GradeBook que usa um array para armazenar notas de teste.
3 // As funções-membro são definidas em GradeBook.cpp
4
5 #include <string> // o programa utiliza a classe string da C++ Standard Library
6 using std::string;
7
8 // Definição da classe GradeBook
9 class GradeBook
10 {
11 public:
12 // constante -- número de alunos que fizeram o teste
13 const static int students = 10; // note os dados públicos
14
15 // construtor inicializa o nome do curso e o array de notas
16 GradeBook(string, const int []);
17
18 void setCourseName(string); // função para configurar o nome do curso
19 string getCourseName(); // função para recuperar o nome do curso
20 void displayMessage(); // exibe uma mensagem de boas-vindas
21 void processGrades(); // realiza várias operações nos dados
22 int getMinimum(); // localiza a nota mínima para o teste
23 int getMaximum(); // localiza a nota máxima para o teste
24 double getAverage(); // determina a nota média para o teste
25 void outputBarChart(); // gera saída do gráfico de barras de distribuição de notas
26 void outputGrades(); // gera a saída do conteúdo do array de notas
27 private:
28 string courseName; // nome do curso para esse livro de notas
29 int grades[students]; // array de notas de aluno
30 }; // fim da classe GradeBook
```

**Figura 7.16** Definição da classe GradeBook utilizando um array para armazenar notas de teste.

```

1 // Figura 7.17: GradeBook.cpp
2 // Definições de função-membro para a classe GradeBook que
3 // utiliza um array para armazenar notas de teste.
4 #include <iostream>
5 using std::cout;
6 using std::cin;
7 using std::endl;
8 using std::fixed;
9
10 #include <iomanip>
11 using std::setprecision;
12 using std::setw;
13
14 #include "GradeBook.h" // definição da classe GradeBook
15
16 // o construtor inicializa courseName e o array grades
17 GradeBook::GradeBook(string name, const int gradesArray[])
18 {
19 setCourseName(name); // inicializa courseName
20
21 // copia notas de gradeArray para membro de dados grades
22 for (int grade = 0; grade < students; grade++)
23 grades[grade] = gradesArray[grade];
24 } // fim do construtor GradeBook
25
26 // função para configurar o nome do curso
27 void GradeBook::setCourseName(string name)
28 {
29 courseName = name; // armazena o nome do curso
30 } // fim da função setCourseName
31
32 // função para recuperar o nome do curso
33 string GradeBook::getCourseName()
34 {
35 return courseName;
36 } // fim da função getCourseName
37
38 // exibe uma mensagem de boas-vindas para o usuário de GradeBook
39 void GradeBook::displayMessage()
40 {
41 // essa instrução chama getCourseName para obter o
42 // nome do curso que esse GradeBook representa
43 cout << "Welcome to the grade book for\n" << getCourseName() << "!"
44 << endl;
45 } // fim da função displayMessage
46
47 // realiza várias operações nos dados
48 void GradeBook::processGrades()
49 {
50 // gera saída de array de notas
51 outputGrades();
52
53 // chama função getAverage para calcular a nota média
54 cout << "\nClass average is " << setprecision(2) << fixed <<
55 getAverage() << endl;
56

```

**Figura 7.17** Funções-membro da classe GradeBook que manipulam um array de notas.

(continua)

```
57 // chama funções getMinimum e getMaximum
58 cout << "Lowest grade is " << getMinimum() << "\nHighest grade is "
59 << getMaximum() << endl;
60
61 // chama outputBarChart para imprimir gráfico de distribuição de notas
62 outputBarChart();
63 } // fim da função processGrades
64
65 // localiza nota mínima
66 int GradeBook::getMinimum()
67 {
68 int lowGrade = 100; // supõe que a nota mais baixa é 100
69
70 // faz um loop pelo array de notas
71 for (int grade = 0; grade < students; grade++)
72 {
73 // se nota for mais baixa que lowGrade, ela é atribuída a lowGrade
74 if (grades[grade] < lowGrade)
75 lowGrade = grades[grade]; // nova nota mais baixa
76 } // fim do for
77
78 return lowGrade; // retorna nota mais baixa
79 } // fim da função getMinimum
80
81 // localiza nota máxima
82 int GradeBook::getMaximum()
83 {
84 int highGrade = 0; // supõe que a nota mais alta é 0
85
86 // faz um loop pelo array de notas
87 for (int grade = 0; grade < students; grade++)
88 {
89 // se a nota atual for mais alta que highGrade, ela é atribuída a highGrade
90 if (grades[grade] > highGrade)
91 highGrade = grades[grade]; // nova nota mais alta
92 } // fim do for
93
94 return highGrade; // retorna nota mais alta
95 } // fim da função getMaximum
96
97 // determina média para o teste
98 double GradeBook::getAverage()
99 {
100 int total = 0; // inicializa o total
101
102 // soma notas no array
103 for (int grade = 0; grade < students; grade++)
104 total += grades[grade];
105
106 // retorna média de notas
107 return static_cast< double >(total) / students;
108 } // fim da função getAverage
109
110 // gera a saída do gráfico de barras exibindo distribuição de notas
111 void GradeBook::outputBarChart()
112 {
```

Figura 7.17 Funções-membro da classe GradeBook que manipulam um array de notas.

(continua)

```

113 cout << "\nGrade distribution:" << endl;
114
115 // armazena frequência de notas em cada intervalo de 10 notas
116 const int frequencySize = 11;
117 int frequency[frequencySize] = { 0 };
118
119 // para cada nota, incrementa a frequência apropriada
120 for (int grade = 0; grade < students; grade++)
121 frequency[grades[grade] / 10]++;
122
123 // para cada frequência de nota, imprime barra no gráfico
124 for (int count = 0; count < frequencySize; count++)
125 {
126 // gera a saída do rótulo das barras ("0-9:", ..., "90-99:", "100:")
127 if (count == 0)
128 cout << " 0-9: ";
129 else if (count == 10)
130 cout << " 100: ";
131 else
132 cout << count * 10 << "- " << (count * 10) + 9 << ": ";
133
134 // imprime a barra de asteriscos
135 for (int stars = 0; stars < frequency[count]; stars++)
136 cout << '*';
137
138 cout << endl; // inicia uma nova linha de saída
139 } // fim do for externo
140 } // fim da função outputBarChart
141
142 // gera a saída do conteúdo do array de notas
143 void GradeBook::outputGrades()
144 {
145 cout << "\nThe grades are:\n\n";
146
147 // gera a saída da nota de cada aluno
148 for (int student = 0; student < students; student++)
149 cout << "Student " << setw(2) << student + 1 << ": " << setw(3)
150 << grades[student] << endl;
151 } // fim da função outputGrades

```

**Figura 7.17** Funções-membro da classe GradeBook que manipulam um array de notas.

(continuação)

Um membro de dados `static` pode ser acessado a partir da definição da classe e das definições de funções-membro exatamente como qualquer outro membro de dados. Como você logo verá, um membro de dados `public static` também pode ser acessado fora da classe, mesmo quando nenhum objeto da classe existe, utilizando o nome de classe seguido pelo operador binário de solução de escopo (`::`) e pelo nome do membro de dados. Você aprenderá mais sobre membros de dados `static` no Capítulo 10.

O construtor da classe (declarado na linha 16 da Figura 7.16 e definido nas linhas 17–24 da Figura 7.17) tem dois parâmetros — o nome do curso e um array de notas. Quando um programa cria um objeto `GradeBook` (por exemplo, a linha 13 de `fig07_18.cpp`), o programa passa um array `int` existente para o construtor, o qual copia os valores no array passado para o membro de dados `grades` (linhas 22–23 da Figura 7.17). Os valores de nota no array passado poderiam ter sido inseridos a partir do usuário ou lidos de um arquivo em disco (como discutido no Capítulo 17, “Processamento de arquivo”). Em nosso programa de teste, simplesmente inicializamos um array com um conjunto de valores de nota (Figura 7.18, linhas 10–11). Uma vez que as notas estão armazenadas no membro de dados `grades` da classe `GradeBook`, todas as funções-membro da classe podem acessar array `grades` conforme necessário para realizar vários cálculos.

A função-membro `processGrades` (declarada na linha 21 da Figura 7.16 e definida nas linhas 48–63 da Figura 7.17) contém uma série de chamadas de função-membro que geram como saída um relatório que resume as notas. A linha 51 chama a função-membro `outputGrades` para imprimir o conteúdo do array `grades`. As linhas 148–150 na função-membro `outputGrades` utiliza uma instrução `for`



para gerar saída de cada nota de aluno. Embora os índices de array iniciem em 0, em geral, um professor numeraria os alunos iniciando em 1. Portanto, as linhas 149–150 geram saída de `student + 1` como o número de aluno para produzir os rótulos de nota "Student 1: ", "Student 2: " e assim por diante.

```

1 // Figura 7.18: fig07_18.cpp
2 // Cria um objeto GradeBook utilizando um array de notas.
3
4 #include "GradeBook.h" // definição da classe GradeBook
5
6 // a função main inicia a execução do programa
7 int main()
8 {
9 // array de notas de aluno
10 int gradesArray[GradeBook::students] =
11 { 87, 68, 94, 100, 83, 78, 85, 91, 76, 87 };
12
13 GradeBook myGradeBook(
14 "CS101 Introduction to C++ Programming", gradesArray);
15 myGradeBook.displayMessage();
16 myGradeBook.processGrades();
17 return 0;
18 } // fim de main

```

```

Welcome to the grade book for
CS101 Introduction to C++ Programming!

```

The grades are:

```

Student 1: 87
Student 2: 68
Student 3: 94
Student 4: 100
Student 5: 83
Student 6: 78
Student 7: 85
Student 8: 91
Student 9: 76
Student 10: 87

```

```

Class average is 84.90
Lowest grade is 68
Highest grade is 100

```

Grade distribution:

```

0-9:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: *
70-79: **
80-89: ****
90-99: **
100: *

```

**Figura 7.18** Cria um objeto GradeBook utilizando um array de notas, e então invoca a função-membro processGrades para analisá-los.

A função-membro `processGrades` em seguida chama a função-membro `getAverage` (linhas 54–55) para obter a média das notas no array. A função-membro `getAverage` (declarada na linha 24 da Figura 7.16 e definida nas linhas 98–108) utiliza uma instrução `for` para somar os valores do array `grades` antes de calcular a média. Observe que o cálculo da média na linha 107 utiliza membro de dados `const static students` para determinar o número de notas cuja média está sendo calculada.

As linhas 58–59 na função-membro `processGrades` chamam as funções-membro `getMinimum` e `getMaximum` para determinar as notas mais baixas e mais altas de qualquer aluno no exame, respectivamente. Vamos examinar como a função-membro `getMinimum` localiza a nota mais baixa. Como a nota mais alta permitida é 100, iniciamos supondo que 100 é a nota mais baixa (linha 68). Então, comparamos cada um dos elementos no array com a nota mais baixa, procurando valores menores. As linhas 71–76 na função-membro `getMinimum` fazem um loop pelo array, e as linhas 74–75 comparam cada nota com `lowGrade`. Se uma nota for menor que `lowGrade`, `lowGrade` é configurado como essa nota. Quando a linha 78 executa, `lowGrade` contém a nota mais baixa no array. A função-membro `getMaximum` (linhas 82–95) funciona de maneira semelhante à função-membro `getMinimum`.

Por fim, a linha 62 na função-membro `processGrades` chama a função-membro `outputBarChart` para imprimir um gráfico de distribuição dos dados de notas utilizando uma técnica semelhante àquela na Figura 7.9. Naquele exemplo, calculamos manualmente o número de notas em cada categoria (isto é, 0–9, 10–19, ..., 90–99 e 100) simplesmente examinando um conjunto de notas. Neste exemplo, as linhas 120–121 utilizam uma técnica semelhante à mostrada nas figuras 7.10 e 7.11 para calcular a frequência de notas em cada categoria. A linha 117 declara e cria o array `frequency` de 11 `ints` para armazenar a frequência de notas em cada categoria de nota. Para cada `grade` no array `grades`, as linhas 120–121 incrementam o elemento apropriado do array `frequency`. Para determinar qual elemento incrementar, a linha 121 divide a `grade` atual por 10 utilizando a divisão de inteiro. Por exemplo, se `grade` for 85, a linha 121 incrementa `frequency[ 8 ]` para atualizar a contagem de notas no intervalo 80–89. As linhas 124–139 a seguir imprimem o gráfico de barras (ver Figura 7.18) com base nos valores no array `frequency`. Como as linhas 29–30 da Figura 7.9, as linhas 135–136 da Figura 7.17 utilizam um valor no array `frequency` para determinar o número de asteriscos a exibir em cada barra.

### Testando a classe *GradeBook*

O programa da Figura 7.18 cria um objeto da classe `GradeBook` (figuras 7.16–7.17) utilizando o array `int gradesArray` (declarado e inicializado nas linhas 10–11). Observe que utilizamos o operador binário de solução de escopo (`::`) na expressão `'GradeBook::students'` (linha 10) para acessar a constante `static students` da classe `GradeBook`. Utilizamos essa constante aqui para criar um array que tem o mesmo tamanho que o array `grades` armazenado como um membro de dados na classe `GradeBook`. As linhas 13–14 passam um nome do curso e o `gradesArray` para o construtor `GradeBook`. A linha 15 exibe uma mensagem de boas-vindas e a linha 16 invoca a função-membro `processGrades` do objeto `GradeBook`. A saída revela o resumo das 10 notas em `myGradeBook`.

## 7.7 Pesquisando arrays com pesquisa linear

Um programador costuma trabalhar com grandes quantidades de dados armazenados em arrays. Pode ser necessário determinar se um array contém um valor que corresponde a um certo **valor-chave**. O processo de localizar um elemento particular de um array é chamado de **pesquisa**. Nesta seção discutimos a pesquisa linear simples. O Exercício 7.33 no final deste capítulo pede para você implementar uma versão recursiva da pesquisa linear. No Capítulo 20, “Pesquisa e classificação”, apresentamos a pesquisa binária, que é mais eficiente, porém mais complexa.

### Pesquisa linear

A **pesquisa linear** (Figura 7.19, linhas 37–44) compara cada elemento de um array com uma **chave de pesquisa** (linha 40). Como o array não está em nenhuma ordem particular, é apenas provável que o valor localizado esteja no primeiro elemento. Em média, portanto, o programa deve comparar a chave de pesquisa com metade dos elementos do array. Para determinar que um valor não está no array, o programa deve comparar a chave de pesquisa com cada elemento no array.

O método de pesquisa linear funciona bem para arrays pequenos ou para arrays não classificados (isto é, arrays cujos elementos não estão em uma ordem particular). Contudo, para arrays grandes, a pesquisa linear é ineficiente. Se o array é classificado (por exemplo, seus elementos estão em ordem crescente), você pode utilizar a técnica de pesquisa binária de alta velocidade sobre a qual aprenderemos no Capítulo 20, “Pesquisa e classificação”.

## 7.8 Classificando arrays por inserção

**Classificar** dados (isto é, colocar os dados em alguma ordem particular como crescente ou decrescente) é uma das aplicações mais importantes da computação. Um banco classifica todos os cheques pelo número de conta de modo que possa preparar extratos bancários individuais no final de cada mês. As empresas de telefonia classificam suas listas telefônicas por sobrenome e, dentro disso, pelo primeiro nome para facilitar a localização de números de telefone. Praticamente todas as organizações devem classificar algum tipo de dados e, em muitos casos, quantidades maciças de dados. Classificar dados é um problema intrigante que tem atraído alguns dos esforços mais intensos de pesquisa no campo de ciência da computação. Neste capítulo, discutimos um esquema simples de classificação. Nos exercícios e no Capítulo 20, “Pesquisa e classificação”, investigamos esquemas mais complexos que resultam em melhor desempenho e introduzimos a notação *O* para caracterizar o grau de dificuldade que cada esquema apresenta para realizar sua tarefa.



### Dica de desempenho 7.4

Às vezes, algoritmos simples fornecem um desempenho pobre. Sua virtude é que eles são fáceis de escrever, testar e depurar. Às vezes são necessários algoritmos mais complexos para alcançar um desempenho ótimo.

```

1 // Figura 7.19: fig07_19.cpp
2 // Pesquisa linear de um array.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 int linearSearch(const int [], int, int); // protótipo
9
10 int main()
11 {
12 const int arraySize = 100; // tamanho do array a
13 int a[arraySize]; // cria o array a
14 int searchKey; // valor a localizar no array a
15
16 for (int i = 0; i < arraySize; i++)
17 a[i] = 2 * i; // cria alguns dados
18
19 cout << "Enter integer search key: ";
20 cin >> searchKey;
21
22 // tenta localizar searchKey no array a
23 int element = linearSearch(a, searchKey, arraySize);
24
25 // exibe os resultados
26 if (element != -1)
27 cout << "Found value in element " << element << endl;
28 else
29 cout << "Value not found" << endl;
30
31 return 0; // indica terminação bem-sucedida
32 } // fim de main
33
34 // compara a chave com cada elemento do array até que a localização seja
35 // encontrada ou até que o fim do array seja alcançado; retorna o subscrito do
36 // elemento se a chave for encontrada ou -1 caso contrário
37 int linearSearch(const int array[], int key, int sizeOfArray)
38 {
39 for (int j = 0; j < sizeOfArray; j++)
40 if (array[j] == key) // se localizado,
41 return j; // retorna a localização da chave
42
43 return -1; // chave não-localizada
44 } // fim da função linearSearch

```

Enter integer search key: 36  
Found value in element 18

Enter integer search key: 37  
Value not found

**Figura 7.19** Pesquisa linear de um array.

*Classificação por inserção*

O programa na Figura 7.20 classifica os valores do array de 10 elementos `data` em ordem crescente. A técnica que utilizamos é chamada **classificação por inserção** — um algoritmo de classificação simples, mas ineficiente. A primeira iteração desse algoritmo pega o segundo elemento e, se ele for menor que o primeiro, troca-o por este (isto é, o programa *insere* o segundo elemento na frente do primeiro elemento). A segunda iteração examina o terceiro elemento e o insere na posição correta com relação aos dois primeiros elementos, de modo que todos os três elementos estejam na ordem. Na  $i$ -ésima iteração desse algoritmo, os primeiros  $i$  elementos no array original estarão classificados.

A linha 13 da Figura 7.20 declara e inicializa o array `data` com os seguintes valores:

34 56 4 10 77 51 93 30 5 52

O programa primeiro consulta `data[ 0 ]` e `data[ 1 ]`, cujos valores são 34 e 56, respectivamente. Esses dois elementos já estão em ordem, então o programa continua — se estivessem fora de ordem, o programa trocava um pelo outro.

Na segunda iteração, o programa consulta o valor de `data[ 2 ]`, 4. Esse valor é menor que 56, então o programa armazena 4 em uma variável temporária e move 56 um elemento para a direita. O programa então verifica e determina que 4 é menor que 34, e move 34 um elemento para a direita. O programa agora alcançou o começo do array, então coloca 4 em `data[ 0 ]`. O array agora está

4 34 56 10 77 51 93 30 5 52

Na terceira iteração, o programa armazena o valor de `data[ 3 ]`, 10, em uma variável temporária. Depois o programa compara 10 com 56 e move 56 um elemento para a direita porque ele é maior que 10. O programa então compara 10 com 34, movendo 34 para a direita um elemento. Quando o programa compara 10 com 4, ele vê que 10 é maior que 4 e coloca 10 em `data[ 1 ]`. O array agora está

4 10 34 56 77 51 93 30 5 52

Utilizando esse algoritmo, na  $i$ -ésima iteração, os primeiros  $i$  elementos do array original são classificados. Entretanto, talvez eles não estejam nas suas localizações finais porque os menores valores podem ser localizados mais tarde no array.

A classificação é realizada pela instrução `for` nas linhas 24–39 que itera pelos elementos do array. A cada iteração, a linha 26 temporariamente armazena na variável `insert` (declarada na linha 14) o valor do elemento que será inserido na parte classificada do array. A linha 28 declara e inicializa a variável `moveItem`, que monitora onde inserir o elemento. As linhas 31–36 fazem um loop para localizar a posição correta onde o elemento deve ser inserido. O loop termina quando o programa alcançar o início do array ou quando alcançar um elemento menor que o valor a ser inserido. A linha 34 move um elemento para a direita, e a linha 35 decrementa a posição em que inserir o próximo elemento. Depois que o loop `while` termina, a linha 38 insere o elemento na posição. Quando a instrução `for` nas linhas 24–39 termina, os elementos do array estão classificados.

A principal virtude da classificação por inserção é que ela é fácil de programar; no entanto, sua execução é lenta. Isso fica evidente quando classificamos arrays grandes. Nos exercícios, investigaremos alguns algoritmos alternativos para classificar um array. Investigamos classificação e pesquisa em profundidade maior no Capítulo 20.

```

1 // Figura 7.20: fig07_20.cpp
2 // Este programa classifica valores de um array em ordem crescente.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 int main()
11 {
12 const int arraySize = 10; // tamanho do array a
13 int data[arraySize] = { 34, 56, 4, 10, 77, 51, 93, 30, 5, 52 };
14 int insert; // variável temporária para armazenar o elemento a inserir
15
16 cout << "Unsorted array:\n";
17
18 // gera saída do array original
19 for (int i = 0; i < arraySize; i++)
20 cout << setw(4) << data[i];
21

```

**Figura 7.20** Classificando um array com classificação por inserção.

(continua)

```

22 // classificação por inserção
23 // itera pelos elementos do array
24 for (int next = 1; next < arraySize; next++)
25 {
26 insert = data[next]; // armazena o valor no elemento atual
27
28 int moveItem = next; // inicializa a localização para colocar elemento
29
30 // procura a localização em que colocar o elemento atual
31 while ((moveItem > 0) && (data[moveItem - 1] > insert))
32 {
33 // desloca o elemento uma posição para a direita
34 data[moveItem] = data[moveItem - 1];
35 moveItem--;
36 } // fim do while
37
38 data[moveItem] = insert; // lugar em que o elemento é inserido no array
39 } // fim do for
40
41 cout << "\nSorted array:\n";
42
43 // gera a saída do array classificado
44 for (int i = 0; i < arraySize; i++)
45 cout << setw(4) << data[i];
46
47 cout << endl;
48 return 0; // indica terminação bem-sucedida
49 } // fim de main

```

Unsorted array:

34 56 4 10 77 51 93 30 5 52

Sorted array:

4 5 10 30 34 51 52 56 77 93

**Figura 7.20** Classificando um array com classificação por inserção.

(continuação)

## 7.9 Arrays multidimensionais

Os **arrays multidimensionais** com duas dimensões costumam ser utilizados para representar **tabelas de valores** consistindo em informações organizadas em **linhas** e **colunas**. Para identificar um elemento particular da tabela, devemos especificar dois subscritos. Por convenção, o primeiro identifica a linha do elemento e o segundo identifica a coluna do elemento. Os arrays que requerem dois subscritos para identificar um elemento particular são chamados de **arrays bidimensionais** ou **arrays 2-D**. Observe que arrays multidimensionais podem ter mais que duas dimensões (por exemplo, subscritos). A Figura 7.21 ilustra um array bidimensional, *a*. O array contém três linhas e quatro colunas, então dizemos que ele é um array de 3 por 4. Em geral, um array com *m* linhas e *n* colunas é chamado de **array *m* por *n***.

Cada elemento no array *a* é identificado na Figura 7.21 por um nome de elemento na forma *a*[ *i* ][ *j* ], onde *a* é o nome do array e *i* e *j* são os subscritos que identificam de maneira única cada elemento em *a*. Observe que todos os nomes dos elementos na linha 0 têm um primeiro subscrito de 0; todos os nomes dos elementos na coluna 3 têm um segundo subscrito de 3.



### Erro comum de programação 7.12

*Referenciar um elemento de array bidimensional *a*[ *x* ][ *y* ] incorretamente como *a*[ *x*, *y* ] é um erro. Na realidade, *a*[ *x*, *y* ] é tratado como *a*[ *y* ], porque o C++ avalia a expressão *x*, *y* (contendo um operador vírgula) simplesmente como *y* (a última das expressões separadas por vírgulas).*

Um array multidimensional pode ser inicializado em sua declaração de maneira muito semelhante a um array unidimensional. Por exemplo, um array bidimensional *b* com os valores 1 e 2 nos elementos da sua linha 0 e os valores 3 e 4 nos elementos da sua linha 1 poderiam ser declarados e inicializados com

```
int b[2][2] = { { 1, 2 }, { 3, 4 } };
```

Os valores são agrupados por linha entre chaves. Então 1 e 2 inicializam `b[ 0 ][ 0 ]` e `b[ 0 ][ 1 ]`, respectivamente, e 3 e 4 inicializam `b[ 1 ][ 0 ]` e `b[ 1 ][ 1 ]`, respectivamente. Se houver inicializadores suficientes para uma dada linha, os elementos restantes dessa linha são inicializados como 0. Portanto, a declaração

```
int b[2][2] = { { 1 }, { 3, 4 } };
```

inicializa `b[ 0 ][ 0 ]` como 1, `b[ 0 ][ 1 ]` como 0, `b[ 1 ][ 0 ]` como 3 e `b[ 1 ][ 1 ]` como 4.

A Figura 7.22 demonstra a inicialização de arrays bidimensionais em declarações. As linhas 11–13 declaram três arrays, cada um com duas linhas e três colunas.

A declaração de `array1` (linha 11) fornece seis inicializadores em duas sublistas. A primeira sublista inicializa a linha 0 do array com os valores 1, 2 e 3; e a segunda sublista inicializa a linha 1 do array com os valores para 4, 5 e 6. Se as chaves em torno de cada sublista forem removidas da lista inicializadora `array1`, o compilador inicializa os elementos da linha 0 seguidos pelos elementos da linha 1, produzindo o mesmo resultado.

A declaração de `array2` (linha 12) fornece somente cinco inicializadores. Os inicializadores são atribuídos à linha 0 e depois à linha 1. Qualquer elemento que não tem um inicializador explícito é inicializado como zero, então `array2[ 1 ][ 2 ]` é inicializado como zero.

A declaração `array3` (linha 13) fornece três inicializadores em duas sublistas. A sublista para a linha 0 inicializa explicitamente os primeiros dois elementos da linha 0 como 1 e 2; o terceiro elemento é implicitamente inicializado como zero. A sublista para a linha 1 inicializa explicitamente o primeiro elemento como 4 e inicializa implicitamente os dois últimos elementos como zero.

O programa chama a função `printArray` para gerar a saída dos elementos de cada array. Observe que a definição de função (linhas 27–38) especifica o parâmetro `const int a[ ][ 3 ]`. Quando uma função recebe um array unidimensional como um argumento, os colchetes do array estão vazios na lista de parâmetros da função. O tamanho da primeira dimensão (isto é, o número de linhas) de um array bidimensional também não é requerido, mas todos os tamanhos subsequentes de dimensão são requeridos. O compilador utiliza esses tamanhos para determinar as localizações na memória de elementos em arrays multidimensionais. Todos os elementos do array são armazenados consecutivamente na memória, independentemente do número de dimensão. Em um array bidimensional, a linha 0 é armazenada na memória seguida pela linha 1. Em um array bidimensional, cada linha é um array unidimensional. Para localizar um elemento em uma linha particular, a função deve conhecer exatamente quantos elementos estão em cada linha para assim poder pular o número adequado de posições da memória quando acessar o array. Portanto, ao acessar `a[ 1 ][ 2 ]`, a função sabe que deve pular três elementos da linha 0 na memória para obter a linha 1. Então, a função acessa o elemento 2 dessa linha.

Muitas manipulações de array comuns utilizam as instruções de repetição `for`. Por exemplo, a seguinte instrução `for` configura todos os elementos na linha 2 do array `a` na Figura 7.21 como zero:

```
for (column = 0; column < 4; column++)
 a[2][column] = 0;
```

A instrução `for` varia apenas o segundo subscrito (isto é, o subscrito de coluna). A instrução `for` anterior é equivalente às seguintes instruções de atribuição:

```
a[2][0] = 0;
a[2][1] = 0;
a[2][2] = 0;
a[2][3] = 0;
```

|         | Coluna 0    | Coluna 1    | Coluna 2    | Coluna 3    |
|---------|-------------|-------------|-------------|-------------|
| Linha 0 | a[ 0 ][ 0 ] | a[ 0 ][ 1 ] | a[ 0 ][ 2 ] | a[ 0 ][ 3 ] |
| Linha 1 | a[ 1 ][ 0 ] | a[ 1 ][ 1 ] | a[ 1 ][ 2 ] | a[ 1 ][ 3 ] |
| Linha 2 | a[ 2 ][ 0 ] | a[ 2 ][ 1 ] | a[ 2 ][ 2 ] | a[ 2 ][ 3 ] |

Subscrito de coluna

Subscrito de linha

Nome do array

**Figura 7.21** O array bidimensional com três linhas e quatro colunas.

```

1 // Figura 7.22: fig07_22.cpp
2 // Inicialização de arrays multidimensionais.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 void printArray(const int a[3]); // protótipo
8
9 int main()
10 {
11 int array1[2][3] = { { 1, 2, 3 }, { 4, 5, 6 } };
12 int array2[2][3] = { 1, 2, 3, 4, 5 };
13 int array3[2][3] = { { 1, 2 }, { 4 } };
14
15 cout << "Values in array1 by row are:" << endl;
16 printArray(array1);
17
18 cout << "\nValues in array2 by row are:" << endl;
19 printArray(array2);
20
21 cout << "\nValues in array3 by row are:" << endl;
22 printArray(array3);
23 return 0; // indica terminação bem-sucedida
24 } // fim de main
25
26 // gera saída do array com duas linhas e três colunas
27 void printArray(const int a[3])
28 {
29 // faz um loop pelas linhas do array
30 for (int i = 0; i < 2; i++)
31 {
32 // faz um loop pelas colunas da linha atual
33 for (int j = 0; j < 3; j++)
34 cout << a[i][j] << ' ';
35
36 cout << endl; // inicia nova linha de saída
37 } // fim do for externo
38 } // fim da função printArray

```

Values in array1 by row are:

1 2 3  
4 5 6

Values in array2 by row are:

1 2 3  
4 5 0

Values in array3 by row are:

1 2 0  
4 0 0

**Figura 7.22** Inicializando arrays multidimensionais.



A seguinte instrução for aninhada soma o total de todos os elementos no array a:

```
total = 0;
for (row = 0; row < 3; row++)
 for (column = 0; column < 4; column++)
 total += a[row][column];
```

A instrução for soma os elementos do array uma linha por vez. A instrução for externa começa configurando row (isto é, o subscrito da linha) como 0, assim os elementos da linha 0 podem ser somados pela instrução for interna. A instrução for externa então incrementa row para 1, assim os elementos da linha 1 podem ser somados. Então, a instrução for externa incrementa row para 2, assim os elementos da linha 2 podem ser somados. Quando a instrução for aninhada termina, total contém a soma de todos os elementos do array.

## 7.10 Estudo de caso: classe GradeBook utilizando um array bidimensional

Na Seção 7.6, apresentamos a classe GradeBook (figuras 7.16–7.17), que utilizou um array unidimensional para armazenar notas de alunos em um único exame. Na maioria dos semestres, os alunos fazem vários exames. É provável que os professores queiram analisar as notas do semestre inteiro, de um único aluno e de toda a classe.

### Armazenando notas de aluno em um array bidimensional na classe *GradeBook*

As figuras 7.23–7.24 contêm uma versão da classe GradeBook que utiliza um array bidimensional grades para armazenar as notas de vários alunos em múltiplos exames. Cada linha do array representa as notas de um único aluno para o curso inteiro e cada coluna representa todas as notas que os alunos tiraram em um exame particular. Um programa-cliente, como fig07\_25.cpp, passa o array como um argumento para o construtor GradeBook. Nesse exemplo, utilizamos um array dez por três contendo as notas de três exames de dez alunos.

```
1 // Figura 7.23: GradeBook.h
2 // Definição da classe GradeBook que utiliza um
3 // array bidimensional para armazenar notas de teste.
4 // As funções-membro são definidas em GradeBook.cpp
5 #include <string> // o programa utiliza a classe string da C++ Standard Library
6 using std::string;
7
8 // definição da classe GradeBook
9 class GradeBook
10 {
11 public:
12 // constantes
13 const static int students = 10; // número de alunos
14 const static int tests = 3; // número de testes
15
16 // o construtor inicializa o nome do curso e o array de notas
17 GradeBook(string, const int [][] tests);
18
19 void setCourseName(string); // função para configurar o nome do curso
20 string getCourseName(); // função para recuperar o nome do curso
21 void displayMessage(); // exibe uma mensagem de boas-vindas
22 void processGrades(); // realiza várias operações nos dados
23 int getMinimum(); // localiza a nota mínima no livro de notas
24 int getMaximum(); // localiza a nota máxima no livro de notas
25 double getAverage(const int [], const int); // encontra a média das notas
26 void outputBarChart(); // gera saída do gráfico de barras de distribuição de notas
27 void outputGrades(); // gera a saída do conteúdo do array de notas
28 private:
29 string courseName; // nome do curso para esse livro de notas
30 int grades[students][tests]; // array bidimensional de notas
31 }; // fim da classe GradeBook
```

**Figura 7.23** Definição da classe GradeBook com um array bidimensional para armazenar notas.

```

1 // Figura 7.24: GradeBook.cpp
2 // Definições de função-membro para a classe GradeBook que
3 // utiliza um array bidimensional para armazenar notas.
4 #include <iostream>
5 using std::cout;
6 using std::cin;
7 using std::endl;
8 using std::fixed;
9
10 #include <iomanip> // manipuladores de fluxo parametrizados
11 using std::setprecision; // configura a precisão da saída numérica
12 using std::setw; // configura a largura de campo
13
14 // inclui a definição da classe GradeBook de GradeBook.h
15 #include "GradeBook.h"
16
17 // o construtor de dois argumentos inicializa courseName e array de notas
18 GradeBook::GradeBook(string name, const int gradesArray[][tests])
19 {
20 setCourseName(name); // inicializa courseName
21
22 // copia notas de gradeArray para grades
23 for (int student = 0; student < students; student++)
24
25 for (int test = 0; test < tests; test++)
26 grades[student][test] = gradesArray[student][test];
27 } // fim do construtor GradeBook de dois argumentos
28
29 // função para configurar o nome do curso
30 void GradeBook::setCourseName(string name)
31 {
32 courseName = name; // armazena o nome do curso
33 } // fim da função setCourseName
34
35 // função para recuperar o nome do curso
36 string GradeBook::getCourseName()
37 {
38 return courseName;
39 } // fim da função getCourseName
40
41 // exibe uma mensagem de boas-vindas para o usuário de GradeBook
42 void GradeBook::displayMessage()
43 {
44 // essa instrução chama getCourseName para obter o
45 // nome do curso que esse GradeBook representa
46 cout << "Welcome to the grade book for\n" << getCourseName() << "!"
47 << endl;
48 } // fim da função displayMessage
49
50 // realiza várias operações nos dados
51 void GradeBook::processGrades()
52 {
53 // gera saída de array de notas
54 outputGrades();
55
56 // chama funções getMinimum e getMaximum

```

**Figura 7.24** Definições de função-membro da classe GradeBook que manipulam um array bidimensional de notas.

(continua)

```

57 cout << "\nLowest grade in the grade book is " << getMinimum()
58 << "\nHighest grade in the grade book is " << getMaximum() << endl;
59
60 // gera saída do gráfico de distribuição de notas de todas as notas em todos os testes
61 outputBarChart();
62 } // fim da função processGrades
63
64 // localiza nota mínima
65 int GradeBook::getMinimum()
66 {
67 int lowGrade = 100; // supõe que a nota mais baixa é 100
68
69 // faz um loop pelas linhas do array de notas
70 for (int student = 0; student < students; student++)
71 {
72 // faz um loop pelas colunas da linha atual
73 for (int test = 0; test < tests; test++)
74 {
75 // se a nota for menor que lowGrade, atribui a nota a lowGrade
76 if (grades[student][test] < lowGrade)
77 lowGrade = grades[student][test]; // nova nota mais baixa
78 } // fim do for interno
79 } // fim do for externo
80
81 return lowGrade; // retorna nota mais baixa
82 } // fim da função getMinimum
83
84 // localiza nota máxima
85 int GradeBook::getMaximum()
86 {
87 int highGrade = 0; // supõe que a nota mais alta é 0
88
89 // faz um loop pelas linhas do array de notas
90 for (int student = 0; student < students; student++)
91 {
92 // faz um loop pelas colunas da linha atual
93 for (int test = 0; test < tests; test++)
94 {
95 // se a nota atual for maior que lowGrade, atribui essa nota a highGrade
96 if (grades[student][test] > highGrade)
97 highGrade = grades[student][test]; // nova nota mais alta
98 } // fim do for interno
99 } // fim do for externo
100
101 return highGrade; // retorna nota mais alta
102 } // fim da função getMaximum
103
104 // determina a média de conjunto particular de notas
105 double GradeBook::getAverage(const int setOfGrades[], const int grades)
106 {
107 int total = 0; // inicializa o total
108
109 // soma notas no array
110 for (int grade = 0; grade < grades; grade++)
111 total += setOfGrades[grade];
112

```

**Figura 7.24** Definições de função-membro da classe GradeBook que manipulam um array bidimensional de notas.

(continua)

```

113 // retorna média de notas
114 return static_cast< double >(total) / grades;
115 } // fim da função getAverage
116
117 // gera a saída do gráfico de barras exibindo distribuição de notas
118 void GradeBook::outputBarChart()
119 {
120 cout << "\nOverall grade distribution:" << endl;
121
122 // armazena frequência de notas em cada intervalo de 10 notas
123 const int frequencySize = 11;
124 int frequency[frequencySize] = { 0 };
125
126 // para cada nota, incrementa a frequência apropriada
127 for (int student = 0; student < students; student++)
128 {
129 for (int test = 0; test < tests; test++)
130 ++frequency[grades[student][test] / 10];
131 }
132
133 // para cada frequência de nota, imprime uma barra no gráfico
134 for (int count = 0; count < frequencySize; count++)
135 {
136 // gera saída do rótulo da barra ("0-9:", ..., "90-99:", "100:")
137 if (count == 0)
138 cout << " 0-9: ";
139 else if (count == 10)
140 cout << " 100: ";
141 else
142 cout << count * 10 << "- " << (count * 10) + 9 << ": ";
143
144 // imprime a barra de asteriscos
145 for (int stars = 0; stars < frequency[count]; stars++)
146 cout << '*';
147
148 cout << endl; // inicia uma nova linha de saída
149 } // fim do for externo
150 } // fim da função outputBarChart
151
152 // gera a saída do conteúdo do array de notas
153 void GradeBook::outputGrades()
154 {
155 cout << "\nThe grades are:\n\n";
156 cout << " "; // alinha títulos de coluna
157
158 // cria um título de coluna para cada um dos testes
159 for (int test = 0; test < tests; test++)
160 cout << "Test " << test + 1 << " ";
161
162 cout << "Average" << endl; // título da coluna de média do aluno
163
164 // cria linhas/colunas de texto que representam notas de array
165 for (int student = 0; student < students; student++)
166 {
167 cout << "Student " << setw(2) << student + 1;
168
169 // gera saída de notas do aluno

```

**Figura 7.24** Definições de função-membro da classe GradeBook que manipulam um array bidimensional de notas.

(continua)

```

169 for (int test = 0; test < tests; test++)
170 cout << setw(8) << grades[student][test];
171
172 // chama a função-membro getAverage para calcular a média do aluno;
173 // passa linha de notas e o valor dos testes como argumentos
174 double average = getAverage(grades[student], tests);
175 cout << setw(9) << setprecision(2) << fixed << average << endl;
176 } // fim do for externo
177 } // fim da função outputGrades

```

**Figura 7.24** Definições de função-membro da classe GradeBook que manipulam um array bidimensional de notas.

(continuação)

Cinco funções-membro (declaradas nas linhas 23–27 da Figura 7.23) realizam manipulações de array para processar as notas. Cada uma dessas funções-membro é semelhante à sua contraparte na versão anterior, baseada em um array unidimensional da classe GradeBook (figuras 7.16–7.17). A função-membro `getMinimum` (definida nas linhas 65–82 da Figura 7.24) determina a nota mais baixa de qualquer aluno no semestre. A função-membro `getMaximum` (definida nas linhas 85–102 da Figura 7.24) determina a nota mais alta de qualquer aluno no semestre. A função-membro `getAverage` (linhas 105–115 da Figura 7.24) determina a média de um aluno particular no semestre. A função-membro `outputBarChart` (linhas 118–149 da Figura 7.24) gera saída de um gráfico de barras da distribuição de todas as notas de aluno no semestre. A função-membro `outputGrades` (linhas 152–177 da Figura 7.24) gera saída do array bidimensional em um formato tabular, junto com a média de cada aluno no semestre.

As funções-membro `getMinimum`, `getMaximum`, `outputBarChart` e `outputGrades` iteram pelo array `grades` utilizando instruções `for` aninhadas. Por exemplo, considere a instrução `for` aninhada na função-membro `getMinimum` (linhas 70–79). A instrução `for` externa começa configurando `student` (isto é, o subscrito de linha) como 0, assim os elementos da linha 0 podem ser comparados com a variável `lowGrade` no corpo da instrução `for` interna. A instrução `for` interna itera pelas notas de uma linha particular e compara cada nota com `lowGrade`. Se uma nota for menor que `lowGrade`, `lowGrade` é configurado como essa nota. A instrução `for` externa então incrementa o subscrito da linha para 1. Os elementos da linha 1 são comparados com a variável `lowGrade`. A instrução `for` externa então incrementa o subscrito de linha para 2, e os elementos da linha 2 são comparados com a variável `lowGrade`. Isso se repete até que todas as linhas de `grades` tenham sido percorridas. Quando a execução da instrução aninhada é concluída, `lowGrade` contém a nota mais baixa no array bidimensional. A função-membro `getMaximum` funciona de maneira semelhante à função-membro `getMinimum`.

A função-membro `outputBarChart` na Figura 7.24 é quase idêntica àquela na Figura 7.17. Entretanto, para gerar saída da distribuição total de notas de um semestre inteiro, a função-membro utiliza uma instrução `for` aninhada (linhas 127–130) para criar o array unidimensional `frequency` com base em todas as notas no array bidimensional. O restante do código em cada uma das funções-membro `outputBarChart` que exibem o gráfico é idêntico.

A função-membro `outputGrades` (linhas 152–177) também utiliza instruções `for` aninhadas para gerar a saída de valores do array `grades`, além da média semestral de cada aluno. A saída na Figura 7.25 mostra o resultado, que é semelhante ao formato tabular de um livro de notas de um professor de física. As linhas 158–159 imprimem os títulos de coluna para cada teste. Utilizamos uma instrução `for` controlada por contador para podermos identificar cada teste com um número. De maneira semelhante, a instrução `for` nas linhas 164–176 primeiro gera a saída de um rótulo de linha utilizando uma variável contadora para identificar cada aluno (linha 166). Embora os índices de array iniciem em 0, observe que as linhas 159 e 166 geram saída de `test + 1` e `student + 1`, respectivamente, para produzir números de teste e de aluno que iniciam em 1 (ver Figura 7.25). A instrução `for` interna nas linhas 169–170 utiliza a variável contadora `student` da instrução `for` externa para fazer um loop por uma linha específica do array `grades` e gerar saída da nota de teste de cada aluno. Por fim, a linha 174 obtém a média do semestre de cada aluno passando a linha atual de `grades` (isto é, `grades[ student ]`) para a função-membro `getAverage`.

```

1 // Figura 7.25: fig07_25.cpp
2 // Cria objeto GradeBook utilizando um array bidimensional de notas.
3
4 #include "GradeBook.h" // Definição da classe GradeBook
5
6 // a função main inicia a execução do programa
7 int main()

```

**Figura 7.25** Cria um objeto GradeBook utilizando um array bidimensional de notas, então invoca a função-membro `processGrades` para analisá-los.

(continua)

```

8 {
9 // array bidimensional de notas de aluno
10 int gradesArray[GradeBook::students][GradeBook::tests] =
11 { { 87, 96, 70 },
12 { 68, 87, 90 },
13 { 94, 100, 90 },
14 { 100, 81, 82 },
15 { 83, 65, 85 },
16 { 78, 87, 65 },
17 { 85, 75, 83 },
18 { 91, 94, 100 },
19 { 76, 72, 84 },
20 { 87, 93, 73 } };
21
22 GradeBook myGradeBook(
23 "CS101 Introduction to C++ Programming", gradesArray);
24 myGradeBook.displayMessage();
25 myGradeBook.processGrades();
26 return 0; // indica terminação bem-sucedida
27 } // fim de main

```

Welcome to the grade book for  
CS101 Introduction to C++ Programming!

The grades are:

|            | Test 1 | Test 2 | Test 3 | Average |
|------------|--------|--------|--------|---------|
| Student 1  | 87     | 96     | 70     | 84.33   |
| Student 2  | 68     | 87     | 90     | 81.67   |
| Student 3  | 94     | 100    | 90     | 94.67   |
| Student 4  | 100    | 81     | 82     | 87.67   |
| Student 5  | 83     | 65     | 85     | 77.67   |
| Student 6  | 78     | 87     | 65     | 76.67   |
| Student 7  | 85     | 75     | 83     | 81.00   |
| Student 8  | 91     | 94     | 100    | 95.00   |
| Student 9  | 76     | 72     | 84     | 77.33   |
| Student 10 | 87     | 93     | 73     | 84.33   |

Lowest grade in the grade book is 65  
Highest grade in the grade book is 100

Overall grade distribution:

```

0-9:
10-19:
20-29:
30-39:
40-49:
50-59:
60-69: ***
70-79: *****
80-89: *****
90-99: *****
100: ***

```

**Figura 7.25** Cria um objeto GradeBook utilizando um array bidimensional de notas, então invoca a função-membro processGrades para analisá-los.

(continuação)

A função-membro `getAverage` (linhas 105–115) aceita dois argumentos — um array unidimensional de resultados de teste para um aluno particular e o número de resultados de teste no array. Quando a linha 174 chama `getAverage`, o primeiro argumento é `grades [ student ]`, que especifica que uma linha particular do array bidimensional `grades` deve ser passada para `getAverage`. Por exemplo, baseado no array criado na Figura 7.25, o argumento `grades [ 1 ]` representa os três valores (um array unidimensional de notas) armazenados na linha 1 do array bidimensional `grades`. Um array bidimensional pode ser considerado um array cujos elementos são arrays de uma dimensão. A função-membro `getAverage` calcula a soma dos elementos do array, divide o total pelo número de resultados do teste e retorna o resultado de ponto flutuante como um valor `double` (linha 114).

### Testando a classe *GradeBook*

O programa na Figura 7.25 cria um objeto da classe `GradeBook` (figuras 7.23–7.24) utilizando o array bidimensional de `ints` chamado `gradesArray` (declarado e inicializado nas linhas 10–20). Observe que a linha 10 acessa as constantes `static students` e `tests` da classe `GradeBook` para indicar o tamanho de cada dimensão do array `gradesArray`. As linhas 22–23 passam um nome do curso e o `gradesArray` para o construtor `GradeBook`. As linhas 24–25 então invocam as funções-membro `displayMessage` e `processGrades` de `myGradeBook` para exibir uma mensagem de boas-vindas e obter um relatório que resume as notas semestrais dos alunos, respectivamente.

## 7.11 Introdução ao template vector da C++ Standard Library

Agora introduzimos o template `vector` da C++ Standard Library, que representa um tipo de array mais robusto que possui muitas capacidades adicionais. Como você verá em capítulos posteriores e em cursos mais avançados de C++, arrays baseados em ponteiro no estilo do C (isto é, o tipo de array apresentado até aqui) têm grande potencial para erros. Por exemplo, como mencionado anteriormente, um programa pode facilmente ‘ultrapassar’ qualquer uma das extremidades de um array, porque o C++ não verifica se os subscritos caem fora do intervalo de um array. Dois arrays não podem ser significativamente comparados com operadores de igualdade nem com operadores relacionais. Como você aprenderá no Capítulo 8, variáveis de ponteiro (conhecidas mais comumente como ponteiros) contêm endereços de memória como seus valores. Os nomes de array são simplesmente ponteiros para onde os arrays iniciam na memória, e, naturalmente, dois arrays sempre estarão em posições de memória diferentes. Quando um array é passado para uma função de uso geral projetada para lidar com arrays de qualquer tamanho, o tamanho do array deve ser passado como um argumento adicional. Além disso, um array não pode ser atribuído a outro com o(s) operador(es) de atribuição — os nomes de array são ponteiros `const` e, como você aprenderá no Capítulo 8, um ponteiro constante não pode ser utilizado no lado esquerdo do operador de atribuição. Essas e outras capacidades certamente parecem ‘naturais’ para lidar com arrays, mas o C++ não fornece tais capacidades. Entretanto, a C++ Standard Library fornece o template de classe `vector` para permitir que os programadores criem uma alternativa mais poderosa e menos propensa a erro para arrays. No Capítulo 11, “Sobrecarga de operadores; objetos `string` e `array`”, apresentamos os modos como implementar essas capacidades de array da maneira como são fornecidas por `vector`. Você aprenderá a personalizar operadores para utilizar com suas próprias classes (uma conhecida técnica de sobrecarga de operador).

O template de classe `vector` está disponível para qualquer pessoa que constrói aplicativos com C++. As notações que o exemplo `vector` utiliza talvez sejam pouco conhecidas para você, porque `vectors` utilizam a notação de template. Lembre-se de que a Seção 6.18 discutiu templates de função. No Capítulo 14, discutimos templates de classe. Por enquanto, você deve se sentir confortável com o template de classe `vector` simulando a sintaxe no exemplo que mostramos nesta seção. Você aprofundará seu entendimento à medida que estudarmos templates de classe no Capítulo 14. O Capítulo 23 apresenta o template de classe `vector` (e várias outras classes contêiner do padrão C++) em detalhes.

O programa da Figura 7.26 demonstra as capacidades fornecidas pelo template `vector` da C++ Standard Library que não estão disponíveis para arrays baseados em ponteiro no estilo do C. O template de classe padrão `vector` fornece muitos dos mesmos recursos que a classe `Array` que construímos no Capítulo 11, “Sobrecarga dos operadores; objetos `string` e `array`”. O template de classe padrão `vector` é definido no cabeçalho `<vector>` (linha 11) e pertence ao namespace `std` (Linha 12). O Capítulo 23 discute a funcionalidade completa do template de classe padrão `vector`.

As linhas 19–20 criam dois objetos `vector` que armazenam valores do tipo `int` — `integers1` contém sete elementos e `integers2` contém 10 elementos. Por padrão, todos os elementos de cada objeto `vector` são configurados como 0. Observe que `vectors` podem ser definidos para armazenar qualquer tipo de dados, substituindo-se `int` em `vector< int >` pelo tipo de dados apropriado. Essa notação, que especifica o tipo armazenado no `vector`, é semelhante à notação de template que a Seção 6.18 introduziu com templates de função. Novamente, o Capítulo 14 discute essa sintaxe em detalhe.

A linha 23 utiliza a função-membro `vector::size` para obter o tamanho (isto é, o número de elementos) de `integers1`. A linha 25 passa `integers1` para a função `outputVector` (linhas 88–102), que utiliza colchetes `[]` para obter o valor em cada elemento do `vector` como um valor que pode ser utilizado para saída. Observe a semelhança dessa notação com a notação utilizada para acessar o valor de um elemento do array. As linhas 28 e 30 realizam as mesmas tarefas para `integers2`.

A função-membro `size` do template de classe `vector` retorna o número de elementos em um `vector` como um valor de tipo `size_t` (que representa o tipo `unsigned int` em muitos sistemas). Como resultado, a linha 90 também declara a variável de controle `i` como do tipo `size_t`. Em alguns compiladores, declarar `i` como um `int` faz com que o compilador emita uma mensagem de advertência, uma vez que a condição de continuação do loop (linha 92) compararia um valor `signed` (isto é, `int i`) e um valor `unsigned` (isto é, um valor do tipo `size_t` retornado pela função `size`).



```

1 // Figura 7.26: fig07_26.cpp
2 // Demonstrando o template de classe vector da C++ Standard Library.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 #include <iomanip>
9 using std::setw;
10
11 #include <vector>
12 using std::vector;
13
14 void outputVector(const vector< int > &); // exibe o vetor
15 void inputVector(vector< int > &); // insere valores no vetor
16
17 int main()
18 {
19 vector< int > integers1(7); // vector< int > de 7 elementos
20 vector< int > integers2(10); // vector< int > de 10 elementos
21
22 // imprime o tamanho e o conteúdo de integers1
23 cout << "Size of vector integers1 is " << integers1.size()
24 << "\nvector after initialization:" << endl;
25 outputVector(integers1);
26
27 // imprime o tamanho e o conteúdo de integers2
28 cout << "\nSize of vector integers2 is " << integers2.size()
29 << "\nvector after initialization:" << endl;
30 outputVector(integers2);
31
32 // insere e imprime integers1 e integers2
33 cout << "\nEnter 17 integers:" << endl;
34 inputVector(integers1);
35 inputVector(integers2);
36
37 cout << "\nAfter input, the vectors contain:\n"
38 << "integers1:" << endl;
39 outputVector(integers1);
40 cout << "integers2:" << endl;
41 outputVector(integers2);
42
43 // utiliza operador de desigualdade (!=) com objetos vector
44 cout << "\nEvaluating: integers1 != integers2" << endl;
45
46 if (integers1 != integers2)
47 cout << "integers1 and integers2 are not equal" << endl;
48
49 // cria o vector integers3 utilizando integers1 como um
50 // inicializador; imprime tamanho e conteúdo
51 vector< int > integers3(integers1); // construtor de cópia
52
53 cout << "\nSize of vector integers3 is " << integers3.size()
54 << "\nvector after initialization:" << endl;
55 outputVector(integers3);
56

```

Figura 7.26 Template vector da C++ Standard Library.

(continua)

```

57 // utiliza operador atribuição (=) sobrecarregado
58 cout << "\nAssigning integers2 to integers1:" << endl;
59 integers1 = integers2; // integers1 é maior que integers2
60
61 cout << "integers1:" << endl;
62 outputVector(integers1);
63 cout << "integers2:" << endl;
64 outputVector(integers2);
65
66 // utiliza operador de igualdade (==) com objetos vector
67 cout << "\nEvaluating: integers1 == integers2" << endl;
68
69 if (integers1 == integers2)
70 cout << "integers1 and integers2 are equal" << endl;
71
72 // utiliza colchetes para criar rvalue
73 cout << "\nintegers1[5] is " << integers1[5];
74
75 // utiliza colchetes para criar lvalue
76 cout << "\n\nAssigning 1000 to integers1[5]" << endl;
77 integers1[5] = 1000;
78 cout << "integers1:" << endl;
79 outputVector(integers1);
80
81 // tentativa de utilizar subscripto fora do intervalo
82 cout << "\nAttempt to assign 1000 to integers1.at(15)" << endl;
83 integers1.at(15) = 1000; // ERRO: fora do intervalo
84 return 0;
85 } // fim de main
86
87 // gera saída do conteúdo do vetor
88 void outputVector(const vector< int > &array)
89 {
90 size_t i; // declara a variável de controle
91
92 for (i = 0; i < array.size(); i++)
93 {
94 cout << setw(12) << array[i];
95
96 if ((i + 1) % 4 == 0) // 4 números por linha de saída
97 cout << endl;
98 } // fim do for
99
100 if (i % 4 != 0)
101 cout << endl;
102 } // fim da função outputVector
103
104 // insere o conteúdo de vetor
105 void inputVector(vector< int > &array)
106 {
107 for (size_t i = 0; i < array.size(); i++)
108 cin >> array[i];
109 } // fim da função inputVector

```

Figura 7.26 Template vector da C++ Standard Library.

(continua)

```

Size of vector integers1 is 7
vector after initialization:
 0 0 0 0
 0 0 0

Size of vector integers2 is 10
vector after initialization:
 0 0 0 0
 0 0 0 0
 0 0

Enter 17 integers:
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

After input, the vectors contain:
integers1:
 1 2 3 4
 5 6 7
integers2:
 8 9 10 11
 12 13 14 15
 16 17

Evaluating: integers1 != integers2
integers1 and integers2 are not equal

Size of vector integers3 is 7
vector after initialization:
 1 2 3 4
 5 6 7

Assigning integers2 to integers1:
integers1:
 8 9 10 11
 12 13 14 15
 16 17
integers2:
 8 9 10 11
 12 13 14 15
 16 17

Evaluating: integers1 == integers2
integers1 and integers2 are equal
integers1[5] is 13

Assigning 1000 to integers1[5]
integers1:
 8 9 10 11
 12 1000 14 15
 16 17

Attempt to assign 1000 to integers1.at(15)

abnormal program termination

```

**Figura 7.26** Template vector da C++ Standard Library.

(continuação)

As linhas 34–35 passam `integers1` e `integers2` à função `inputVector` (linhas 105–109) para ler os valores dos elementos de cada vector fornecidos pelo usuário. A função `inputVector` utiliza colchetes (`[]`) para obter *lvalues* que podem ser utilizados para armazenar os valores de entrada em cada elemento do vector.

A linha 46 demonstra que objetos `vector` podem ser comparados diretamente com o operador `!=`. Se os conteúdos de dois vectors não forem iguais, o operador retorna `true`; caso contrário, o operador retorna `false`.

O template `vector` da C++ Standard Library permite que os programadores criem um novo objeto `vector` que é inicializado com o conteúdo de um vector existente. A linha 51 cria um objeto `vector` (`integers3`) e o inicializa com uma cópia de `integers1`. Isso invoca o chamado construtor de cópia de `vector` para realizar a operação de cópia. Você aprenderá sobre construtores de cópia em detalhe no Capítulo 11. As linhas 53 e 55 geram saída do tamanho e conteúdo de `integers3` para demonstrar que ele foi inicializado corretamente.

A linha 59 atribui `integers2` a `integers1` para demonstrar que o operador de atribuição (`=`) pode ser utilizado com objetos `vector`. As linhas 62 e 64 geram saída do conteúdo de ambos os objetos para mostrar que eles agora contêm valores idênticos. A linha 69 então compara `integers1` com `integers2` utilizando o operador de igualdade (`==`) para determinar se os conteúdos dos dois objetos são iguais depois da atribuição na linha 59 (e eles são).

As linhas 73 e 77 demonstram que um programa pode utilizar colchetes (`[]`) para obter um elemento `vector` como um *lvalue não modificável* e como um *lvalue modificável*, respectivamente. Um *lvalue não modificável* é uma expressão que identifica um objeto na memória (como um elemento em um vector), mas não pode ser utilizado para modificar esse objeto. Um *lvalue modificável* também identifica um objeto na memória, mas pode ser utilizado para modificar o objeto. Como é o caso com arrays baseados em ponteiro no estilo do C, o C++ não realiza qualquer verificação de limites quando elementos `vector` são acessados com colchetes. Portanto, o programador deve assegurar que operações que utilizam `[]` acidentalmente não tentem manipular elementos fora dos limites do vector. O template de classe padrão `vector`, porém, na realidade fornece limite para essa verificação em sua função-membro `at`, que ‘lança uma exceção’ (consulte o Capítulo 16, “Tratamento de exceções”) se seu argumento é um subscrito inválido. Por padrão, isso faz com que um programa C++ termine. Se o subscrito for válido, a função `at` retorna o elemento na localização especificada como um *lvalue modificável* ou um *lvalue não modificável*, dependendo do contexto (não-const ou const) em que a chamada aparece. A linha 83 demonstra uma chamada à função `at` com um subscrito inválido.

Nesta seção, demonstramos o template `vector` da C++ Standard Library, uma classe reutilizável e robusta que pode substituir arrays baseados em ponteiro no estilo do C. No Capítulo 11, você verá que `vector` alcança muitas de suas capacidades ‘sobrecarregando’ operadores predefinidos do C++ e aprenderá a personalizar operadores para utilizar com suas próprias classes de maneiras semelhantes. Por exemplo, criamos uma classe `Array` que, como o template da classe `vector`, aprimora as capacidades básicas de array. Nossa classe `Array` também fornece recursos adicionais, como a capacidade de realizar entrada e saída de arrays inteiros com operadores `>>` e `<<`, respectivamente.

## 7.12 Estudo de caso de engenharia de software: colaboração entre objetos no sistema ATM (opcional)

Nesta seção, nós nos concentramos nas colaborações (interações) entre objetos em nosso sistema de ATM. Quando dois objetos se comunicam para realizar uma tarefa, diz-se que eles são *colaboradores* — os objetos fazem isso invocando as operações um do outro. Uma *colaboração* consiste em um objeto de uma classe enviar uma *mensagem* para um objeto de outra classe. As mensagens são enviadas em C++ via chamadas de função-membro.

Na Seção 6.18, determinamos muitas das operações das classes em nosso sistema. Nesta seção, focalizamos as mensagens que invocam essas operações. Para identificar as colaborações no sistema, retornamos ao documento de requisitos da Seção 2.8. Lembre-se de que esse documento especifica a série de atividades que ocorre durante uma sessão ATM (por exemplo, autenticar um usuário, realizar transações). Os passos utilizados para descrever como o sistema deve realizar cada uma dessas tarefas são nossa primeira indicação das colaborações em nosso sistema. À medida que avançamos por esta e pelas seções restantes do “Estudo de caso de engenharia de software”, podemos descobrir colaborações adicionais.

### Identificando as colaborações em um sistema

Identificamos as colaborações no sistema lendo cuidadosamente as seções do documento de requisitos que especificam o que o ATM deve fazer para autenticar um usuário e realizar todo tipo de transação. Para cada ação ou passo descrito no documento de requisitos, decidimos quais objetos em nosso sistema devem interagir para alcançar o resultado desejado. Identificamos um objeto como o objeto emissor (isto é, o objeto que envia a mensagem) e outro como o objeto receptor (isto é, o objeto que oferece essa operação para clientes da classe). Então selecionamos uma das operações do objeto receptor (identificadas na Seção 6.18) que devem ser invocadas pelo objeto emissor para produzir o comportamento adequado. Por exemplo, o ATM exibe uma mensagem de boas-vindas quando desocupado. Sabemos que um objeto da classe `Screen` exibe uma mensagem para o usuário via sua operação `displayMessage`. Portanto, decidimos que o sistema pode exibir uma mensagem de boas-vindas empregando uma colaboração entre ATM e `Screen` em que ATM envia uma mensagem `displayMessage` para `Screen` invocando a operação `displayMessage` da classe `Screen`. [Nota: Para evitar repetir a frase ‘um objeto da classe...’, nós nos referimos a cada objeto simplesmente utilizando seu nome de classe precedido (ou não) por um artigo (‘um’/‘uma’ ou ‘o’/‘a’) — por exemplo, ‘o ATM’ refere-se a um objeto da classe ATM.]

A Figura 7.27 lista as colaborações que podem ser derivadas do documento de requisitos. Para cada objeto emissor, listamos as colaborações na ordem em que elas são discutidas no documento de requisitos. Listamos cada colaboração envolvendo um único remetente, uma única mensagem e um único destinatário somente uma vez, mesmo que a colaboração possa ocorrer várias vezes durante uma sessão de ATM. Por exemplo, a primeira linha na Figura 7.27 indica que o ATM colabora com o Screen sempre que o ATM precisar exibir uma mensagem para o usuário.

Vamos considerar as colaborações na Figura 7.27. Antes de permitir que um usuário realize qualquer transação, o ATM deve exibir um prompt pedindo para o usuário inserir um número de conta e um PIN. Ele realiza cada uma dessas tarefas enviando uma mensagem `displayMessage` para `Screen`. Essas duas ações referem-se à mesma colaboração entre o ATM e a `Screen`, que já está listada na Figura 7.27. O ATM obtém a entrada em resposta a um prompt enviando uma mensagem `getInput` para `Keypad`. Em seguida, o ATM deve determinar se o número da conta especificado pelo usuário e o PIN correspondem àqueles de uma conta no banco de dados. Ele faz isso enviando uma mensagem `authenticateUser` para `BankDatabase`. Lembre-se de que `BankDatabase` não pode autenticar um usuário diretamente — somente o `Account` do usuário (isto é, o `Account` que contém o número da conta especificado pelo usuário) pode acessar o PIN do usuário para autenticar o usuário. Portanto, a Figura 7.27 lista uma colaboração em que `BankDatabase` envia uma mensagem `validatePIN` para uma `Account`.

Depois que o usuário é autenticado, o ATM exibe o menu principal enviando uma série de mensagens `displayMessage` para a `Screen` e obtém entrada contendo uma seleção de menu enviando uma mensagem `getInput` ao `Keypad`. Nós já explicamos essas colaborações. Depois que o usuário escolher o tipo de transação, o ATM executa a transação enviando uma mensagem `execute` para um objeto da classe de transação apropriada (isto é, um `BalanceInquiry`, um `Withdrawal` ou um `Deposit`). Por exemplo, se o usuário escolher realizar uma consulta de saldo, o ATM envia uma mensagem `execute` para um `BalanceInquiry`.

O exame adicional do documento de requisitos revela as colaborações envolvidas na execução de cada tipo de transação. Um `BalanceInquiry` recupera a quantia de dinheiro disponível na conta do usuário enviando uma mensagem `getAvailableBalance` para `BankDatabase`, que responde enviando uma mensagem `getAvailableBalance` ao `Account` do usuário. De maneira semelhante, `BalanceInquiry` recupera o valor do depósito enviando uma mensagem `getTotalBalance` para `BankDatabase`, que envia a mesma mensagem ao `Account` do usuário. Para exibir os dois tipos de saldo do usuário ao mesmo tempo, `BalanceInquiry` envia uma mensagem `displayMessage` para `Screen`.

`Withdrawal` envia uma série de mensagens `displayMessage` para a `Screen` exibir um menu de valores de saque-padrão (isto é, \$ 20, \$ 40, \$ 60, \$ 100, \$ 200). `Withdrawal` envia uma mensagem `getInput` para `Keypad` obter a seleção de menu do usuário. Em seguida, `Withdrawal` determina se o valor do saque solicitado é menor que ou igual ao saldo da conta do usuário. `Withdrawal` pode obter o

| Um objeto da classe...      | envia a mensagem...                                                                                                                                                                   | para um objeto da classe...                                                                                                                                      |
|-----------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ATM                         | <code>displayMessage</code><br><code>getInput</code><br><code>authenticateUser</code><br><code>execute</code><br><code>execute</code><br><code>execute</code>                         | <code>Screen</code><br><code>Keypad</code><br><code>BankDatabase</code><br><code>BalanceInquiry</code><br><code>Withdrawal</code><br><code>Deposit</code>        |
| <code>BalanceInquiry</code> | <code>getAvailableBalance</code><br><code>getTotalBalance</code><br><code>displayMessage</code>                                                                                       | <code>BankDatabase</code><br><code>BankDatabase</code><br><code>Screen</code>                                                                                    |
| <code>Withdrawal</code>     | <code>displayMessage</code><br><code>getInput</code><br><code>getAvailableBalance</code><br><code>isSufficientCashAvailable</code><br><code>debit</code><br><code>dispenseCash</code> | <code>Screen</code><br><code>Keypad</code><br><code>BankDatabase</code><br><code>CashDispenser</code><br><code>BankDatabase</code><br><code>CashDispenser</code> |
| <code>Deposit</code>        | <code>displayMessage</code><br><code>getInput</code><br><code>isEnvelopeReceived</code><br><code>credit</code>                                                                        | <code>Screen</code><br><code>Keypad</code><br><code>DepositSlot</code><br><code>BankDatabase</code>                                                              |
| <code>BankDatabase</code>   | <code>validatePIN</code><br><code>getAvailableBalance</code><br><code>getTotalBalance</code><br><code>debit</code><br><code>credit</code>                                             | <code>Account</code><br><code>Account</code><br><code>Account</code><br><code>Account</code><br><code>Account</code>                                             |

**Figura 7.27** Colaborações no sistema ATM.

valor disponível na conta do usuário enviando uma mensagem `getAvailableBalance` para `BankDatabase`. `Withdrawal` então testa se o dispensador de cédulas contém dinheiro suficiente enviando uma mensagem `isSufficientCashAvailable` para `CashDispenser`. `Withdrawal` envia uma mensagem `debit` para `BankDatabase` para diminuir o saldo da conta do usuário. O `BankDatabase`, por sua vez, envia a mesma mensagem para a `Account` apropriada. Lembre-se de que debitar fundos de uma `Account` diminui tanto `totalBalance` como `availableBalance`. Para liberar o valor solicitado, `Withdrawal` envia uma mensagem `dispenseCash` para `CashDispenser`. Por fim, `Withdrawal` envia uma mensagem `displayMessage` para `Screen`, que instrui o usuário a pegar o dinheiro.

Um `Deposit` responde a uma mensagem `execute` enviando primeiro uma mensagem `displayMessage` para `Screen` para solicitar ao usuário o valor do depósito. `Deposit` envia uma mensagem `getInput` para `Keypad` para obter a entrada do usuário. `Deposit` então envia uma mensagem `displayMessage` para `Screen` para instruir o usuário a inserir um envelope de depósito. Para determinar se a abertura de depósito recebeu um envelope de depósito, `Deposit` envia uma mensagem `isEnvelopeReceived` para `DepositSlot`. `Deposit` atualiza a conta do usuário enviando uma mensagem `credit` para `BankDatabase`, que subsequentemente envia uma mensagem `credit` à `Account` do usuário. Lembre-se de que creditar fundos em uma `Account` aumenta `totalBalance` mas não o `availableBalance`.

### Diagramas de interação

Agora que identificamos um conjunto de colaborações possíveis entre os objetos em nosso sistema ATM, vamos modelar graficamente essas interações utilizando a UML. A UML fornece vários tipos de **diagramas de interação** que modelam o comportamento de um sistema modelando a maneira como os objetos interagem entre si. O **diagrama de comunicação** enfatiza os objetos que participam das colaborações. [Nota: Os diagramas de comunicação foram chamados **diagramas de colaboração** nas versões anteriores da UML.] Como o diagrama de comunicação, o **diagrama de seqüências** mostra as colaborações entre objetos, mas enfatiza *quando* as mensagens são enviadas entre objetos *ao longo do tempo*.

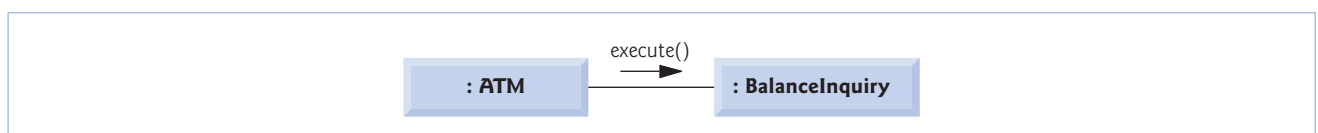
### Diagramas de comunicação

A Figura 7.28 mostra um diagrama de comunicação que modela o ATM para executar um `BalanceInquiry`. Os objetos são modelados na UML como retângulos que contêm nomes na forma `nomeDoObjeto : NomeDaClasse`. Nesse exemplo, que envolve apenas um objeto de cada tipo, não damos importância ao nome de objeto e listamos apenas dois-pontos seguidos pelo nome da classe. [Nota: Ao modelar múltiplos objetos do mesmo tipo, recomenda-se especificar o nome de cada objeto em um diagrama de comunicação.] Os objetos que se comunicam são conectados com linhas sólidas e as mensagens são passadas entre os objetos ao longo dessas linhas na direção mostrada por setas. O nome da mensagem, que aparece junto à seta, é o nome de uma operação (isto é, uma função-membro) que pertence ao objeto receptor — considere o nome como um serviço que o objeto receptor fornece para enviar objetos (seus ‘clientes’).

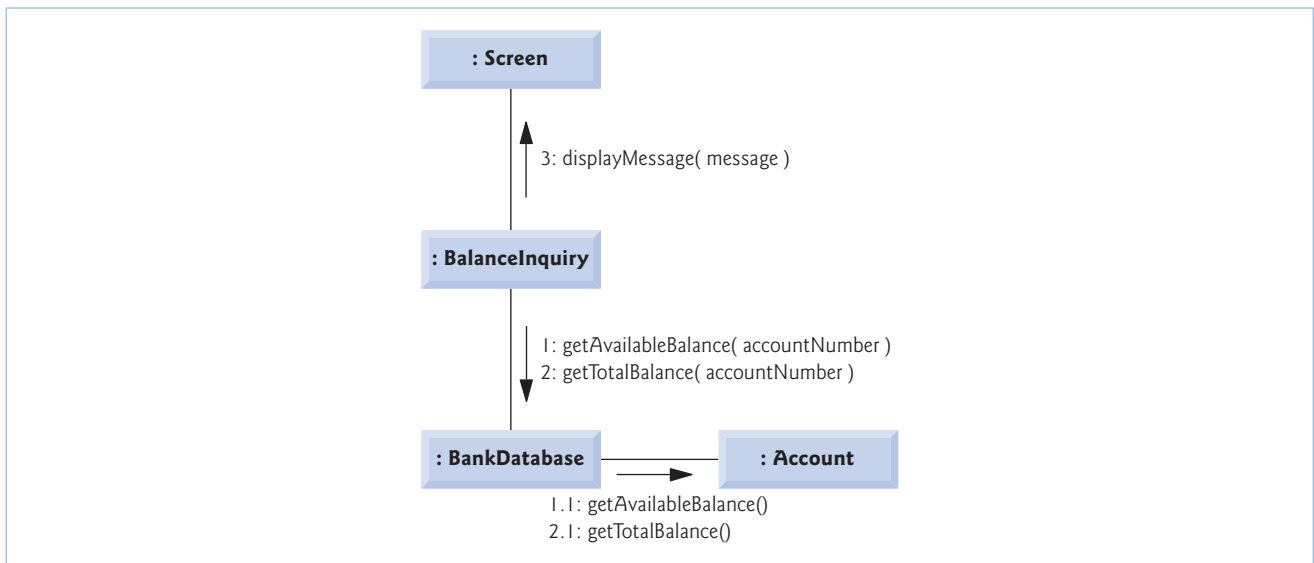
A seta de preenchimento sólido na Figura 7.28 representa uma mensagem — ou **chamada síncrona** — na UML e uma chamada de função em C++. Essa seta indica que o fluxo de controle é do objeto emissor (o ATM) para o objeto receptor (um `BalanceInquiry`). Visto que essa é uma chamada síncrona, o objeto emissor pode não enviar outra mensagem, ou não fazer coisa alguma, até que o objeto receptor processe a mensagem e retorne o controle para o objeto emissor. O emissor simplesmente espera. Por exemplo, na Figura 7.28, o ATM chama a função-membro `execute` de um `BalanceInquiry` e pode não enviar outra mensagem até que `execute` tenha concluído e retornado o controle para o ATM. [Nota: Se isso fosse uma **chamada assíncrona**, representada por uma seta, o objeto emissor não teria de esperar o objeto receptor para retornar o controle — ele continuaria enviando mensagens adicionais imediatamente após a chamada assíncrona. Chamadas assíncronas freqüentemente podem ser implementadas em C++ utilizando bibliotecas específicas de plataforma fornecidas com seu compilador. Tais técnicas estão além do escopo deste livro.]

### Seqüência de mensagens em um diagrama de comunicação

A Figura 7.29 mostra um diagrama de comunicação que modela as interações entre objetos no sistema quando um objeto da classe `BalanceInquiry` é executado. Supomos que o atributo `accountNumber` do objeto contém o número da conta do usuário atual. As colaborações na Figura 7.29 iniciam depois de o ATM enviar uma mensagem `execute` para um `BalanceInquiry` (isto é, a interação modelada na Figura 7.28). O número à esquerda de um nome de mensagem indica a ordem em que a mensagem é passada. A **seqüência de mensagens** em um diagrama de comunicação progride em ordem numérica do menor para o maior. Nesse diagrama, a numeração inicia com a mensagem 1 e termina com a mensagem 3. O `BalanceInquiry` primeiro envia uma mensagem `getAvailableBalance` para o `BankDatabase` (mensagem 1), então envia uma mensagem `getTotalBalance` para o `BankDatabase` (mensagem 2). Dentro dos parênteses que se seguem a um nome de mensagem, podemos especificar uma lista separada por vírgulas dos nomes dos parâmetros enviados com a mensagem (isto é, argumentos em uma chamada de função em C++) — o atributo `BalanceInquiry` passa `accountNumber` com suas mensagens para o `BankDatabase` para indicar quais informações de saldo de `Account` recuperar. A partir da Figura 6.33, lembre-se de que as operações `getAvailableBalance` e `getTotalBalance` da classe `BankDatabase` exigem ambas um parâmetro para



**Figura 7.28** Diagrama de comunicação do ATM que executa uma consulta de saldo.



**Figura 7.29** Diagrama de comunicação para executar uma consulta de saldo.

identificar uma conta. O próximo `BalanceInquiry` exibe `availableBalance` e `totalBalance` para o usuário, passando uma mensagem `displayMessage` para `Screen` (mensagem 3) que inclui um parâmetro que indica `message` a ser exibida.

Note, porém, que a Figura 7.29 modela duas mensagens adicionais que passam do `BankDatabase` para uma `Account` (mensagem 1.1 e mensagem 2.1). Para fornecer ao ATM os dois saldos da `Account` do usuário (como solicitado pelas mensagens 1 e 2), `BankDatabase` deve passar um `getAvailableBalance` e uma mensagem `getTotalBalance` para `Account` do usuário. Essas mensagens passadas dentro do tratamento de outra mensagem são chamadas de **mensagens aninhadas**. A UML recomenda utilizar um esquema de numeração decimal para indicar mensagens aninhadas. Por exemplo, a mensagem 1.1 é a primeira mensagem aninhada na mensagem 1 — `BankDatabase` passa uma mensagem `getAvailableBalance` durante o processamento por `BankDatabase` de uma mensagem com o mesmo nome. [Nota: Se `BankDatabase` precisasse passar uma segunda mensagem aninhada durante o processamento da mensagem 1, a segunda mensagem seria numerada 1.2.] Uma mensagem só pode ser passada quando todas as mensagens aninhadas da mensagem anterior forem passadas. Por exemplo, `BalanceInquiry` passa a mensagem 3 somente depois que as mensagens 2 e 2.1 forem passadas, nessa ordem.

O esquema aninhado de numeração utilizado nos diagramas de comunicação ajuda a esclarecer precisamente quando e em que contexto cada mensagem é passada. Por exemplo, se numerássemos as mensagens na Figura 7.29 utilizando um esquema de numeração simples (isto é, 1, 2, 3, 4, 5), alguém examinando o diagrama poderia não ser capaz de determinar que `BankDatabase` passa a mensagem `getAvailableBalance` (mensagem 1.1) para um `Account` *durante* o processamento por `BankDatabase` da mensagem 1, em oposição a *depois* de completar o processamento da mensagem 1. Os números decimais aninhados deixam claro que a segunda mensagem `getAvailableBalance` (mensagem 1.1) é passada para um `Account` dentro do tratamento da primeira mensagem `getAvailableBalance` (mensagem 1) pelo `BankDatabase`.

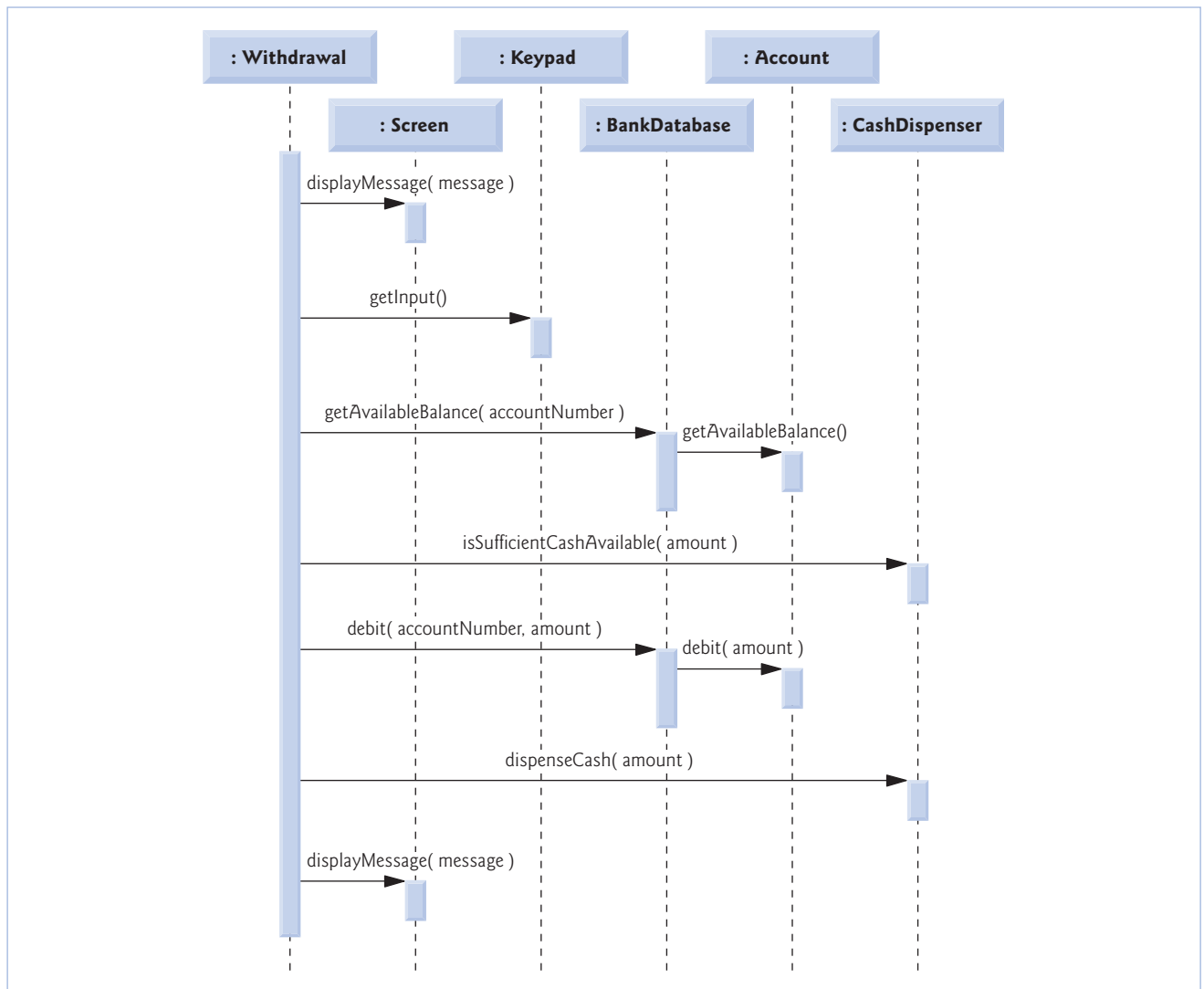
### Diagramas de seqüências

Os diagramas de comunicação enfatizam os participantes de colaborações, mas modelam sua sincronização de uma maneira um pouco complicada. Um diagrama de seqüências ajuda a modelar a sincronização de colaborações mais claramente. A Figura 7.30 mostra um diagrama de seqüências modelando a seqüência de interações que ocorre quando um `Withdrawal` executa. A linha pontilhada que se estende para baixo do retângulo de um objeto é a **linha da vida** desse objeto, que representa a progressão de tempo. As ações geralmente ocorrem ao longo da linha da vida de um objeto na ordem cronológica de cima para baixo — uma ação próxima da parte superior acontece antes de uma próxima da parte inferior.

A passagem de mensagem em diagramas de seqüências é semelhante à passagem de mensagem em diagramas de comunicação. Uma seta sólida com uma seta preenchida que se estende do objeto emissor para o objeto receptor representa uma mensagem entre dois objetos. A seta aponta para uma ativação na linha da vida do objeto receptor. Uma **ativação**, mostrada como um retângulo vertical estreito, indica que um objeto está executando. Quando um objeto retorna o controle, uma mensagem de retorno, representada como uma linha tracejada com uma seta, estende-se desde a ativação do objeto que está retornando o controle até a ativação do objeto que inicialmente enviou a mensagem. Para eliminar a poluição visual, omitimos as setas de retorno de mensagem — a UML permite essa prática para tornar os diagramas mais legíveis. Como os diagramas de comunicação, os diagramas de seqüências podem indicar parâmetros de mensagem entre os parênteses que se seguem a um nome de mensagem.

A seqüência de mensagens na Figura 7.30 inicia quando um `Withdrawal` solicita ao usuário que escolha um valor de saque enviando uma mensagem `displayMessage` para `Screen`. `Withdrawal` então envia uma mensagem `getInput` para `Keypad`, que obtém a entrada do usuário. Já modelamos a lógica de controle envolvida em um `Withdrawal` no diagrama de atividades da Figura 5.28, então não mostra-





**Figura 7.30** Diagrama de seqüências que modela um `Withdrawal` em execução.

mos essa lógica no diagrama de seqüências da Figura 7.30. Em vez disso, modelamos o cenário mais favorável em que o saldo da conta do usuário é maior que ou igual ao valor retirado escolhido e o dispensador de cédulas contém uma quantia suficiente de dinheiro para atender à solicitação. Para informações sobre como modelar a lógica de controle em um diagrama de seqüências, consulte os recursos da Web e as leituras recomendadas listadas no fim da Seção 2.8.

Depois de obter o valor de um saque, `Withdrawal` envia uma mensagem `getAvailableBalance` para `BankDatabase`, que por sua vez envia uma mensagem `getAvailableBalance` para a `Account` do usuário. Supondo que a conta do usuário tem dinheiro suficiente disponível para permitir a transação, `Withdrawal` em seguida envia uma mensagem `isSufficientCashAvailable` para o `CashDispenser`. Supondo que há dinheiro suficiente disponível, `Withdrawal` diminui o saldo da conta do usuário (isto é, tanto de `totalBalance` como de `availableBalance`) enviando uma mensagem `debit` para `BankDatabase`. `BankDatabase` responde enviando uma mensagem `debit` para o `Account` do usuário. Por fim, `Withdrawal` envia uma mensagem `dispenseCash` para `CashDispenser` e uma mensagem `displayMessage` para `Screen`, que pede para o usuário retirar o dinheiro da máquina.

Identificamos as colaborações entre objetos no sistema ATM e modelamos algumas dessas colaborações utilizando diagramas de interação UML — tanto diagramas de comunicação como diagramas de seqüências. Na próxima seção de “Estudo de caso de engenharia de software” (Seção 9.12), aprimoramos a estrutura de nosso modelo para completar um projeto orientado a objetos preliminar, então começamos a implementação do sistema ATM.

### Exercícios de revisão do estudo de caso de engenharia de software

- 7.1** Um(a) \_\_\_\_\_ consiste em um objeto de uma classe que envia uma mensagem para um objeto de outra classe.
- a) associação
  - b) agregação
  - c) colaboração
  - d) composição

- 7.2** Que forma de diagrama de interação enfatiza *quais* colaborações ocorrem? Que forma enfatiza *quando* as colaborações ocorrem?
- 7.3** Crie um diagrama de seqüências que modela as interações entre os objetos no sistema ATM que ocorrem quando um `Deposit` executa com sucesso e explique a seqüência de mensagens modeladas pelo diagrama.

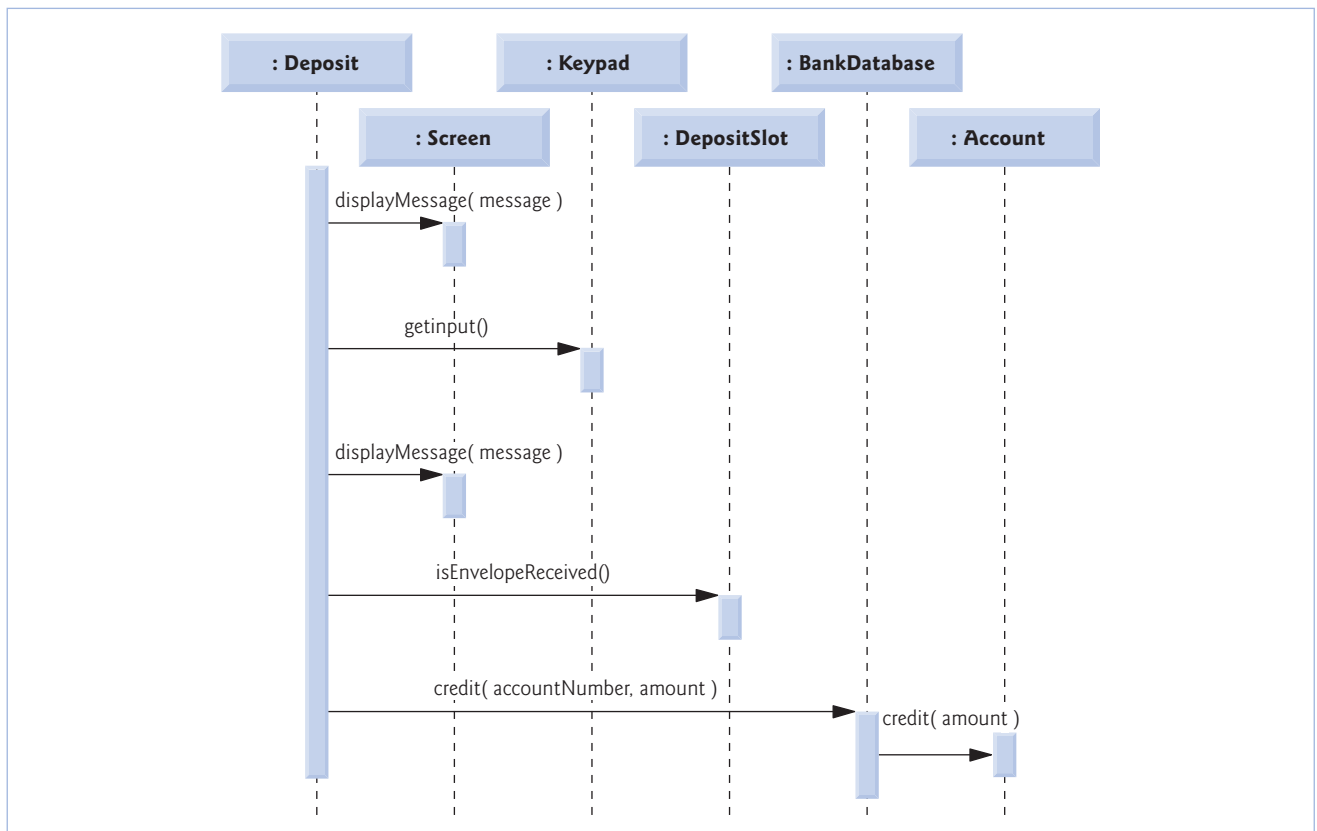
### Respostas aos exercícios de revisão do estudo de caso de engenharia de software

- 7.1** c.
- 7.2** Os diagramas de comunicação enfatizam *quais* colaborações ocorrem. Os diagramas de seqüências enfatizam *quando* as colaborações ocorrem.
- 7.3** A Figura 7.31 apresenta um diagrama de seqüências que modela as interações entre objetos no sistema ATM que ocorrem quando um `Deposit` executa com sucesso. A Figura 7.31 indica que um `Deposit` primeiro envia uma mensagem `displayMessage` a `Screen` para pedir ao usuário que insira o valor do depósito. Em seguida `Deposit` envia uma mensagem `getInput` a `Keypad` para receber a entrada do usuário. `Deposit` então instrui o usuário a inserir um envelope de depósito enviando uma mensagem `displayMessage` a `Screen`. `Deposit` em seguida envia uma mensagem `isEnvelopeReceived` a `DepositSlot` para confirmar que o envelope de depósito foi recebido pela ATM. Por fim, `Deposit` aumenta o atributo `totalBalance` (mas não o atributo `availableBalance`) do `Account` do usuário enviando uma mensagem `credit` para `BankDatabase`. `BankDatabase` responde enviando a mesma mensagem para o `Account` do usuário.

## 7.13 Síntese

Este capítulo iniciou nossa introdução às estruturas de dados, explorando o uso de arrays e vectors para armazenar dados e recuperar dados de listas e tabelas de valores. Os exemplos do capítulo demonstraram como declarar um array, inicializar um array e referenciar elementos individuais de um array. Também ilustramos como passar arrays a funções e como utilizar o qualificador `const` para impor o princípio do menor privilégio. Os exemplos do capítulo também apresentaram técnicas de pesquisa de classificação básicas. Você aprendeu a declarar e manipular arrays multidimensionais. Por fim, demonstramos a capacidade do template vector da C++ Standard Library, que fornece uma alternativa mais robusta aos arrays.

Continuamos nossa cobertura de estruturas de dados no Capítulo 14, “Templates”, no qual construímos um template de classe de pilha, e no Capítulo 21, “Estruturas de dados”, que introduz estruturas de dados dinâmicas, como listas, filas, pilhas e árvores, que podem crescer e encolher à medida que os programas executam. O Capítulo 23, “Standard Template Library (STL)”, introduz várias das



**Figura 7.31** Diagrama de seqüências que modela um `Deposit` em execução.

estruturas de dados predefinidas da C++ Standard Library, as quais os programadores podem utilizar em vez de construir suas próprias. O Capítulo 23 apresenta a funcionalidade completa do template da classe `vector` e discute muitas outras classes de estrutura de dados, incluindo `list` e `deque`, que são estruturas de dados do tipo array que podem crescer e encolher em resposta à mudança dos requisitos de armazenamento de um programa.

Agora introduzimos os conceitos básicos de classes, objetos, instruções de controle, funções e arrays. No Capítulo 8, apresentamos um dos recursos mais poderosos do C++ — o ponteiro. Os ponteiros monitoram onde dados e funções são armazenados na memória, o que nos permite manipular esses itens de maneiras interessantes. Depois de introduzir conceitos básicos de ponteiro, examinamos em detalhe o íntimo relacionamento entre arrays, ponteiros e strings.

## Resumo

- As estruturas de dados são coleções de itens de dados relacionados. Arrays são estruturas de dados consistindo em itens de dados do mesmo tipo relacionados. Os arrays são entidades ‘estáticas’ no sentido de que permanecem com o mesmo tamanho ao longo de toda a execução do programa. (Naturalmente, elas podem ser de uma classe de armazenamento automático e daí serem criadas e destruídas toda vez que o fluxo do programa entra e sai dos blocos em que elas são definidas.)
- Um array é um grupo consecutivo de posições da memória que compartilham o mesmo tipo.
- Para referir-se a uma localização ou elemento particular em um array, especificamos o nome do array e o número de posição do elemento particular no array.
- Um programa referencia qualquer um dos elementos de um array dando o nome do array seguido pelo número de posição do elemento particular entre colchetes (`[]`). O número de posição mais formalmente é chamado subscrito ou índice. (Esse número especifica o número de elementos a partir do início do array.)
- O primeiro elemento em cada array tem subscrito zero e às vezes é chamado de zero-ésimo elemento.
- Um subscrito deve ser um inteiro ou uma expressão do tipo inteiro (que utiliza qualquer tipo inteiro).
- É importante notar a diferença entre o ‘sétimo elemento do array’ e o ‘elemento 7 do array’. Subscritos de array iniciam em 0, portanto o ‘sétimo elemento do array’ tem um subscrito de 6, enquanto o ‘elemento 7 do array’ tem um subscrito de 7 e na realidade é o oitavo elemento do array. Essa distinção costuma ser uma fonte de erros *off-by-one*.
- Os colchetes utilizados para incluir o subscrito de um array são na realidade um operador em C++. Os colchetes têm o mesmo nível de precedência que os parênteses.
- Os arrays ocupam espaço na memória. O programador especifica o tipo de cada elemento e o número de elementos requeridos por um array como segue:

```
tipo nomeDoArray[tamanhoDoArray];
```

e o compilador reserva a quantidade apropriada de memória.

- Arrays podem ser declarados para conter qualquer tipo de dados. Por exemplo, um array do tipo `char` pode ser utilizado para armazenar uma string de caracteres.
- Os elementos de um array podem ser inicializados na declaração do array colocando depois do nome de array um sinal de igual e uma lista inicializadora — uma lista separada por vírgulas (incluída entre chaves) de constantes inicializadores. Ao inicializar um array com uma lista inicializadora, se houver menos inicializadores que elementos no array, os elementos restantes são inicializados como zero.
- Se o tamanho do array for omitido de uma declaração com uma lista inicializadora, o compilador determina o número de elementos no array contando o número de elementos na lista inicializadora.
- Se o tamanho do array e uma lista inicializadora forem especificados em uma declaração de array, o número de inicializadores deve ser menor que ou igual ao tamanho do array. Fornecer mais inicializadores em uma lista inicializadora de array do que o número de elementos existentes no array é um erro de compilação.
- Constantes devem ser inicializadas com uma expressão constante ao serem declaradas e não podem ser modificadas depois. Constantes podem ser colocadas em qualquer lugar em que uma expressão constante é esperada.
- O C++ não tem nenhuma verificação de limites de array para evitar que o computador reference um elemento que não existe. Portanto, um programa que executa pode ‘ultrapassar’ qualquer uma das extremidades de um array sem aviso. Os programadores devem assegurar que todas as referências de array permaneçam dentro dos limites do array.
- Um array de caracteres pode ser inicializado utilizando um literal string. O tamanho de um array de caracteres é determinado pelo compilador com base no comprimento da string *mais* um caractere especial de terminação de string chamado caractere nulo (representado pela constante de caractere `'\0'`).
- Todas as strings representadas por arrays de caracteres acabam com o caractere nulo. Um array de caracteres representando uma string sempre deve ser declarado com um tamanho grande o suficiente para armazenar o número de caracteres na string e o caractere nulo de terminação.
- Os arrays de caracteres também podem ser inicializados com constantes de caractere individuais em uma lista inicializadora.

- Caracteres individuais em uma string podem ser acessados diretamente com a notação de subscrito de array.
- Uma string pode ser inserida diretamente em um array de caracteres a partir do teclado utilizando `cin` e `>>`.
- Um array de caracteres representando uma string terminada por caractere nulo pode ser enviado para a saída com `cout` e `<<`.
- Uma variável local `static` em uma definição de função existe até o fim do programa, mas é visível somente no corpo da função.
- Um programa inicializa arrays locais `static` quando suas declarações são encontradas pela primeira vez. Se um array `static` não é inicializado explicitamente pelo programador, cada elemento desse array é inicializado como zero pelo compilador quando o array é criado.
- Para passar um argumento de array a uma função, especifique o nome do array sem colchetes. Para passar um elemento de um array para uma função, utilize o nome subscrito do elemento do array como um argumento na chamada de função.
- Os arrays são passados a funções por referência — as funções chamadas podem modificar o valor dos elementos nos arrays originais dos chamadores. O valor do nome do array é o endereço na memória do computador do primeiro elemento do array. Como o endereço inicial do array é passado, a função chamada sabe precisamente onde o array está armazenado na memória.
- Elementos individuais de um array são passados por valor exatamente como variáveis simples o são. Esses fragmentos de dados simples são chamados escalares ou quantidades escalares.
- Para receber um argumento de array, a lista de parâmetros da função deve especificar que a função espera receber um array. O tamanho do array não é requerido entre os colchetes do array.
- O C++ fornece o qualificador de tipo `const` que pode ser utilizado para evitar modificação de valores do array no chamador por meio de código em uma função chamada. Quando um parâmetro de array é precedido pelo qualificador `const`, os elementos do array tornam-se constantes no corpo da função e qualquer tentativa de modificar um elemento do array no corpo da função resulta em um erro de compilação.
- A pesquisa linear compara cada elemento de um array com uma chave de pesquisa. Como o array não está em nenhuma ordem particular, é apenas provável que o valor localizado esteja no primeiro elemento. Em média, portanto, um programa deve comparar a chave de pesquisa com metade dos elementos do array. Para determinar que um valor não está no array, o programa deve comparar a chave de pesquisa com cada elemento no array. O método de pesquisa linear funciona bem para arrays pequenos e é aceitável para arrays não classificados.
- Um array pode ser classificado utilizando a classificação por inserção. A primeira iteração desse algoritmo pega o segundo elemento e, se ele for menor que o primeiro, troca-o por este (isto é, o programa *insere* o segundo elemento na frente do primeiro elemento). A segunda iteração examina o terceiro elemento e o insere na posição correta com relação aos dois primeiros elementos, de modo que todos os três elementos estejam na ordem. Na  $i$ -ésima iteração desse algoritmo, os primeiros  $i$  elementos no array original estarão classificados. Para arrays pequenos, a classificação por inserção é aceitável, mas para arrays maiores é ineficiente comparada com outros algoritmos de classificação mais sofisticados.
- Os arrays multidimensionais com duas dimensões costumam ser utilizados para representar tabelas de valores consistindo em informações organizadas em linhas e colunas.
- Os arrays que requerem dois subscritos para identificar um elemento particular são chamados arrays bidimensionais. Um array com  $m$  linhas e  $n$  colunas é chamado de array  $m$  por  $n$ .
- O template `vector` da C++ Standard Library representa uma alternativa mais robusta aos arrays por apresentar muitas capacidades que não são oferecidas pelos arrays baseados em ponteiro no estilo do C.
- Por padrão, todos os elementos de um objeto `vector` inteiro são configurados como 0.
- Um `vector` pode ser definido para armazenar qualquer tipo de dados utilizando uma declaração na forma:

```
vector< tipo > nome(tamanho);
```

- A função-membro `size` da template da classe `vector` retorna o número de elementos no `vector` em que é invocada.
- O valor de um elemento de um `vector` pode ser acessado ou modificado utilizando colchetes (`[]`).
- Os objetos do template de classe padrão `vector` podem ser comparados diretamente com os operadores de igualdade (`==`) e desigualdade (`!=`). O operador de atribuição (`=`) também pode ser utilizado com objetos `vector`.
- Um *lvalue* não modificável é uma expressão que identifica um objeto na memória (como um elemento em um vetor), mas não pode ser utilizado para modificar esse objeto. Um *lvalue* modificável também identifica um objeto na memória, mas pode ser utilizado para modificar o objeto.
- O template de classe padrão `vector` fornece verificação de limites em sua função-membro `at`, que 'lança uma exceção' se seu argumento for um subscrito inválido. Por padrão, isso faz com que um programa C++ termine.

## Terminologia

array 2-D

`a[ i ]`

`a[ i ][ j ]`

array

array bidimensional

array  $m$  por  $n$

array multidimensional

array unidimensional

`at`, função-membro de `vector`

caractere nulo (`'\0'`)

chave de pesquisa

classificação por inserção

classificar um array

colchetes `[]`

coluna de um array bidimensional

`const`, qualificador de tipo

constante identificada

declarar um array

|                                 |                                                       |                                                        |
|---------------------------------|-------------------------------------------------------|--------------------------------------------------------|
| elemento de um array            | <i>lvalue</i> modificável                             | string representada por um array de caracteres         |
| erro, <i>off-by-one</i>         | <i>lvalue</i> não modificável                         | subscrito                                              |
| escalabilidade                  | nome de um array                                      | subscrito de coluna                                    |
| escalar                         | número de posição                                     | subscrito de linha                                     |
| estrutura de dados              | número mágico                                         | tabela de valores                                      |
| formato tabular                 | passando arrays a funções                             | 'ultrapassar' os limites de um array                   |
| índice                          | passar por referência                                 | valor de um elemento                                   |
| índice zero                     | pesquisa linear de um array                           | valor-chave                                            |
| inicializador                   | pesquisar um array                                    | variáveis de leitura                                   |
| inicializar um array            | quantidade escalar                                    | variável constante                                     |
| linha de um array bidimensional | <code>size</code> , função-membro <code>vector</code> | <code>vector</code> (template da C++ Standard Library) |
| lista inicializadora            | <code>static</code> , membro de dados                 | verificação de limites                                 |
| lista inicializadora de array   | string representada por um array de caracteres        | zero-ésimo elemento                                    |

## Exercícios de revisão

**7.1** Complete cada uma das seguintes sentenças:

- Listas e tabelas de valores podem ser armazenadas em \_\_\_\_\_ ou \_\_\_\_\_.
- Os elementos de um array são relacionados pelo fato de eles terem o mesmo \_\_\_\_\_ e \_\_\_\_\_.
- O número utilizado para referenciar um elemento particular de um array é chamado de seu \_\_\_\_\_.
- Um(a) \_\_\_\_\_ deve ser utilizado(a) para declarar o tamanho de um array, porque torna o programa mais escalonável.
- O processo de colocar os elementos de um array em ordem é chamado de \_\_\_\_\_ o array.
- O processo de determinar se um array contém um valor-chave particular é chamado de \_\_\_\_\_ o array.
- Um array que utiliza dois subscritos é referido como um array \_\_\_\_\_.

**7.2** Determine se as seguintes sentenças são *verdadeiras* ou *falsas*. Se a resposta for *falsa*, explicar por quê.

- Um array pode armazenar muitos tipos de valores diferentes.
- Um subscrito de array normalmente deve ser do tipo de dados `float`.
- Se houver menos inicializadores em uma lista inicializadora que o número de elementos no array, os elementos restantes são inicializados com o último valor na lista inicializadora.
- É um erro se uma lista inicializadora contiver mais inicializadores que o número de elementos no array.
- Um elemento do array individual que é passado para uma função e modificado nessa função conterá o valor modificado quando a função chamada completar sua execução.

**7.3** Escreva uma ou mais instruções que realizam as seguintes tarefas para um array chamado `fractions`:

- Defina uma variável constante `arraySize` inicializada como 10.
- Declare um array com `arraySize` elementos do tipo `double` e inicialize os elementos como 0.
- Nomeie o quarto elemento do array.
- Referencie o elemento 4 do array.
- Atribua o valor 1.667 ao elemento 9 do array.
- Atribua o valor 3.333 ao sétimo elemento do array.
- Imprima os elementos do array 6 e 9 com dois dígitos de precisão à direita do ponto de fração decimal e mostre a saída que realmente é exibida na tela.
- Imprima todos os elementos do array utilizando uma instrução `for`. Defina a variável de inteiro `i` como uma variável de controle para o loop. Mostre a saída.

**7.4** Responda às seguintes perguntas relacionadas a um array chamado `table`:

- Declare o array como um array de inteiros e tendo 3 linhas e 3 colunas. Suponha que a variável constante `arraySize` tenha sido definida como 3.
- Quantos elementos o array contém?
- Utilize uma instrução de repetição `for` para inicializar cada elemento do array com a soma de seus subscritos. Suponha que as variáveis inteiras `i` e `j` são declaradas como variáveis de controle.
- Escreva um segmento de programa para imprimir os valores de cada elemento do array `table` em formato tabular com 3 linhas e 3 colunas. Suponha que o array foi inicializado com a declaração

```
int table[arraySize][arraySize] = { { 1, 8 }, { 2, 4, 6 }, { 5 } };
```

e que as variáveis inteiras `i` e `j` são declaradas como variáveis de controle. Mostre a saída.

**7.5** Localize o erro em cada um dos seguintes segmentos de programa e corrija-o:

- `#include <iostream>;`

- b) `arraySize = 10; // arraySize foi declarado const`  
 c) Suponha que `int b[ 10 ] = { 0 };`  
     `for ( int i = 0; i <= 10; i++ )`  
         `b[ i ] = 1;`  
 d) Suponha que `int a[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };`  
     `a[ 1, 1 ] = 5;`

## Respostas dos exercícios de revisão

- 7.1** a) arrays, vectors. b) nome, tipo. c) subscrito (ou índice). d) variável constante. e) classificar. f) pesquisar. g) bidimensional.

- 7.2** a) Falsa. Um array pode armazenar apenas valores do mesmo tipo.  
 b) Falsa. Um subscrito de array deve ser um inteiro ou uma expressão do tipo inteiro.  
 c) Falsa. Os elementos restantes são inicializados como zero.  
 d) Verdadeira.  
 e) Falsa. Elementos individuais de um array são passados por valor. Se o array inteiro é passado para uma função, então quaisquer modificações serão refletidas no original.

- 7.3** a) `const int arraySize = 10;`  
 b) `double fractions[ arraySize ] = { 0.0 };`  
 c) `fractions[ 3 ]`  
 d) `fractions[ 4 ]`  
 e) `fractions[ 9 ] = 1.667;`  
 f) `fractions[ 6 ] = 3.333;`  
 g) `cout << fixed << setprecision( 2 );`  
     `cout << fractions[ 6 ] << ' ' << fractions[ 9 ] << endl;`  
     Saída: 3.33 1.67.  
 h) `for ( int i = 0; i < arraySize; i++ )`  
     `cout << "fractions[" << i << "] = " << fractions[ i ] << endl;`  
     Saída:  
     `fractions[ 0 ] = 0.0`  
     `fractions[ 1 ] = 0.0`  
     `fractions[ 2 ] = 0.0`  
     `fractions[ 3 ] = 0.0`  
     `fractions[ 4 ] = 0.0`  
     `fractions[ 5 ] = 0.0`  
     `fractions[ 6 ] = 3.333`  
     `fractions[ 7 ] = 0.0`  
     `fractions[ 8 ] = 0.0`  
     `fractions[ 9 ] = 1.667`

- 7.4** a) `int table[ arraySize ][ arraySize ];`  
 b) Nove.  
 c) `for ( i = 0; i < arraySize; i++ )`  
     `for ( j = 0; j < arraySize; j++ )`  
         `table[ i ][ j ] = i + j;`  
 d) `cout << "        [0]   [1]   [2]" << endl;`  
     `for ( int i = 0; i < arraySize; i++ ) {`  
         `cout << '[' << i << "]" << " ";`  
     `for ( int j = 0; j < arraySize; j++ )`  
         `cout << setw( 3 ) << table[ i ][ j ] << " ";`  
     `cout << endl;`

Saída:

```
[0] [1] [2]
[0] 1 8 0
```

```
[1] 2 4 6
[2] 5 0 0
```

- 7.5**
- Erro: O ponto-e-vírgula no final da diretiva de pré-processador `#include`.  
Correção: Elimine o ponto-e-vírgula.
  - Erro: Atribuir um valor a uma variável constante utilizando uma instrução de atribuição.  
Correção: Inicialize a variável constante em uma declaração `const int arraySize`.
  - Erro: Referenciando um elemento de array além dos limites do array (`b[10]`).  
Correção: Altere o valor final da variável de controle para 9.
  - Erro: Atribuição de subscrito de array feita incorretamente.  
Correção: Altere a instrução para `a[ 1 ][ 1 ] = 5;`

## Exercícios

- 7.6** Preencha as lacunas em cada um das seguintes sentenças:
- Os nomes dos quatro elementos do array `p (int p[ 4 ];` são \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_ e \_\_\_\_\_.
  - Nomear um array, declarar seu tipo e especificar o número de dimensões no array é chamado de \_\_\_\_\_ o array.
  - Por convenção, o primeiro subscrito em um array bidimensional identifica um elemento \_\_\_\_\_ e o segundo subscrito identifica um elemento \_\_\_\_\_.
  - Um array *m* por *n* contém \_\_\_\_\_ linhas, \_\_\_\_\_ colunas e \_\_\_\_\_ elementos.
  - O nome do elemento na linha 3 e na coluna 5 do array `d` é \_\_\_\_\_.
- 7.7** Determine se cada uma das seguintes sentenças é *verdadeira* ou *falsa*. Se *falsa*, explique por quê.
- Para referir-se a uma localização particular ou elemento dentro de um array, especificamos o nome do array e o valor do elemento particular.
  - Uma declaração de array reserva espaço para o array.
  - Para indicar que 100 localizações devem ser reservadas para o array de inteiros `p`, o programador escreve a declaração  
`p[ 100 ];`
  - Uma instrução `for` deve ser utilizada para inicializar os elementos de um array de 15 elementos como zero.
  - Instruções `for` aninhadas devem ser utilizadas para somar os elementos de um array bidimensional.
- 7.8** Escreva instruções C++ para realizar cada uma das seguintes tarefas:
- Exiba o valor do elemento 6 do array de caracteres `f`.
  - Insira um valor no elemento 4 do array unidimensional de ponto flutuante `b`.
  - Inicialize cada um dos 5 elementos do array de inteiros unidimensional `g` como 8.
  - Some e imprima os elementos do array de ponto flutuante `c` de 100 elementos.
  - Copie o array `a` para a primeira parte do array `b`. Considere `double a[ 11 ], b[ 34 ];`
  - Determine e imprima os maiores e menores valores contidos no array de ponto flutuante `w` de 9 elementos.
- 7.9** Considere um array de inteiros 2 por 3 `t`.
- Escreva uma declaração para `t`.
  - Quantas linhas tem `t`?
  - Quantas colunas tem `t`?
  - Quantos elementos tem `t`?
  - Escreva os nomes de todos os elementos na linha 1 de `t`.
  - Escreva os nomes de todos os elementos na coluna 2 de `t`.
  - Escreva uma única instrução que configura o elemento de `t` na linha 1 e na coluna 2 como zero.
  - Escreva uma série de instruções que inicializam cada elemento de `t` como zero. Não utilize um `loop`.
  - Escreva uma instrução `for` aninhada que inicializa cada elemento de `t` como zero.
  - Escreva uma instrução que insere os valores para os elementos de `t` a partir do terminal.
  - Escreva uma série de instruções que determinam e imprimem o menor valor no array `t`.
  - Escreva uma instrução que exibe os elementos na linha 0 de `t`.
  - Escreva uma instrução que soma os elementos na coluna 3 de `t`.
  - Escreva uma série de instruções que imprime o array `t` no formato tabular. Liste os subscritos de coluna como títulos ao longo do topo da tabela e liste os subscritos de linha à esquerda de cada linha.
- 7.10** Utilize um array unidimensional para resolver o seguinte problema. Uma empresa paga seu pessoal de vendas por comissão. Os vendedores recebem \$ 200 por semana mais 9% de suas vendas brutas por semana. Por exemplo, um vendedor que vende \$ 5.000 brutos em uma semana recebe \$ 200 mais 9% de \$ 5.000 ou um total de \$ 650. Escreva um programa (utilizando um array de contadores) que determina quanto o pessoal de vendas ganhou em cada um dos seguintes intervalos (suponha que o salário de cada vendedor foi truncado para uma quantidade do tipo inteiro) :



- a) \$ 200–\$ 299
- b) \$ 300–\$ 399
- c) \$ 400–\$ 499
- d) \$ 500–\$ 599
- e) \$ 600–\$ 699
- f) \$ 700–\$ 799
- g) \$ 800–\$ 899
- h) \$ 900–\$ 999
- i) \$ 1.000 e acima

- 7.11** (*Classificação por borbulhamento (bubble sort)*) No **algoritmo de classificação por borbulhamento**, valores menores gradualmente sobem para a parte superior do array como bolhas de ar subindo na água, enquanto as bolhas maiores afundam. A classificação por borbulhamento faz várias passagens pelo array. Em cada passagem, sucessivos pares de elementos são comparados. Se um par estiver na ordem crescente (ou os valores forem idênticos), deixamos os valores como eles estão. Se um par estiver na ordem decrescente, seus valores são trocados no array. Escreva um programa que classifica um array de 10 inteiros utilizando classificação por borbulhamento.
- 7.12** A classificação por borbulhamento descrita no Exercício 7.11 é ineficiente para arrays grandes. Faça as seguintes modificações simples para aprimorar o desempenho da classificação por borbulhamento:
- a) Depois da primeira passagem, garanta-se que o maior número está no elemento de número mais alto do array; após a segunda passagem, os dois números mais altos estão ‘no lugar’; e assim por diante. Em vez de fazer nove comparações em cada passagem, modifique a classificação por borbulhamento para fazer oito comparações na segunda passagem, sete na terceira passagem e assim por diante.
  - b) Os dados no array já podem estar na ordem adequada ou ordem quase adequada, então por que fazer nove passagens se menos seriam suficientes? Modifique a classificação para verificar no fim de cada passagem se alguma troca foi feita. Se nenhuma troca tiver sido feita, então os dados já devem estar na ordem apropriada; portanto, o programa deve terminar. Se trocas foram feitas, então pelo menos mais uma passagem é necessária.
- 7.13** Escreva instruções simples que realizam as seguintes operações de um array unidimensional:
- a) Inicialize os 10 elementos do array de inteiros `counts` como zeros.
  - b) Adicione 1 a cada um dos 15 elementos do array de inteiros `bonus`.
  - c) Leia 12 valores para o array `double monthlyTemperatures` a partir do teclado.
  - d) Imprima os 5 valores do array de inteiros `bestScores` em formato de coluna.
- 7.14** Localize o(s) erro(s) em cada uma das seguintes instruções:
- a) Suponha que: `char str[ 5 ]`;  

```
cin >> str; // usuário digita "hello"
```
  - b) Suponha que: `int a[ 3 ]`;  

```
cout << a[1] << " " << a[2] << " " << a[3] << endl;
```
  - c) `double f[ 3 ] = { 1.1, 10.01, 100.001, 1000.0001 };`
  - d) Suponha que: `double d[ 2 ][ 10 ]`;  

```
d[1, 9] = 2.345;
```
- 7.15** Utilize um array unidimensional para resolver o seguinte problema. Leia 20 números, cada um dos quais está entre 10 e 100, inclusive. À medida que cada número é lido, valide-o e armazene-o no array somente se ele não for uma duplicata de um número já lido. Depois de ler todos os valores, exiba somente os valores únicos que o usuário inseriu. Previna-se para o ‘pior caso’ em que todos os 20 números são diferentes. Utilize o menor array possível para resolver esse problema.
- 7.16** Rotule os elementos de um array unidimensional de 3 por 5 chamado `sales` para indicar a ordem em que eles são configurados como zero pelo seguinte segmento de programa:
- ```
for ( row = 0; row < 3; row++ )
    for ( column = 0; column < 5; column++ )
        sales[ row ][ column ] = 0;
```
- 7.17** Escreva um programa que simula a rolagem de dois dados. O programa deve utilizar `rand` para rolar o primeiro dado e deve utilizar `rand` novamente para rolar o segundo dado. A soma dos dois valores deve então ser calculada. [Nota: Cada dado pode mostrar um valor inteiro de 1 a 6, portanto a soma dos dois valores variará de 2 a 12, com 7 sendo a soma mais freqüente, e 2 e 12 sendo as somas menos freqüentes. A Figura 7.32 mostra as 36 possíveis combinações de dois dados. Seu programa deve rolar os dois dados 36.000 vezes. Utilize um array unidimensional para contar o número de vezes que cada possível soma aparece. Imprima os resultados em um formato tabular. Além disso, determinará se os totais são razoáveis (isto é, há seis maneiras de rolar um 7, então aproximadamente um sexto de todas as rolagens deve ser 7).]

	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	4	5	6	7	8
3	4	5	6	7	8	9
4	5	6	7	8	9	10
5	6	7	8	9	10	11
6	7	8	9	10	11	12

Figura 7.32 Os 36 possíveis resultados de rolar dois dados.

7.18 O que o seguinte programa faz?

```

1 // Ex. 7.18: Ex07_18.cpp
2 // O que esse programa faz?
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int whatIsThis( int [], int ); // protótipo de função
8
9 int main()
10 {
11     const int arraySize = 10;
12     int a[ arraySize ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
13
14     int result = whatIsThis( a, arraySize );
15
16     cout << "Result is " << result << endl;
17     return 0; // indica terminação bem-sucedida
18 } // fim de main
19
20 // O que essa função faz?
21 int whatIsThis( int b[], int size )
22 {
23     if ( size == 1 ) // caso básico
24         return b[ 0 ];
25     else // passo recursivo
26         return b[ size - 1 ] + whatIsThis( b, size - 1 );
27 } // fim da função whatIsThis

```

7.19 Modifique o programa da Figura 6.11 para jogar 1000 partidas do jogo de dados *craps*. O programa deve monitorar as estatísticas e responder às seguintes perguntas:

- Quantas partidas são ganhas na primeira rolagem dos dados, na segunda, ..., na vigésima e depois da vigésima rolagem dos dados?
- Quantas partidas são perdidas na primeira rolagem dos dados, na segunda, ..., na vigésima e depois da vigésima rolagem dos dados?
- Quais são as chances de ganhar no jogo de dados? [Nota: Você deve descobrir que o *craps* é um dos jogos mais comuns de cassino. O que você supõe que isso significa?]
- Qual é o comprimento médio de um jogo de dados *craps*?
- As chances de ganhar aumentam com o comprimento do jogo?

7.20 (Sistema de reservas de passagens aéreas) Uma pequena companhia aérea acabou de comprar um computador para seu novo sistema automatizado de reservas. Você foi solicitado a programar o novo sistema. Você escreverá um programa para atribuir assentos em cada voo da companhia aérea (capacidade: 10 assentos).

Seu programa deve exibir o seguinte menu de alternativas — Please type 1 for "First Class" e Please type 2 for "Economy". Se a pessoa digitar 1, seu programa deve atribuir um assento na primeira classe (assentos 1-5). Se a pessoa digitar 2, seu programa

deve atribuir um assento na classe econômica (assentos 6-10). Seu programa deve imprimir um bilhete de embarque indicando o número do assento da pessoa e se ela está na primeira classe ou na classe econômica do avião.

Utilize um array unidimensional para representar o gráfico de assentos do avião. Inicialize todos os elementos do array como 0 para indicar que todos os assentos estão vazios. À medida que cada assento é atribuído, configure os elementos correspondentes do array como 1 para indicar que o assento não está mais disponível.

Naturalmente, seu programa nunca deve atribuir um assento que já foi atribuído. Quando a primeira classe estiver lotada, seu programa deve perguntar à pessoa se ela aceita ficar na classe econômica (e vice-versa). Se ela aceitar, faça a atribuição apropriada de assento. Caso contrário, imprima a mensagem "Next flight leaves in 3 hours."

7.21 O que o seguinte programa faz?

```

1  // Ex. 7.21: Ex07_21.cpp
2  // O que esse programa faz?
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  void someFunction( int [], int, int ); // protótipo de função
8
9  int main()
10 {
11     const int arraySize = 10;
12     int a[ arraySize ] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
13
14     cout << "The values in the array are:" << endl;
15     someFunction( a, 0, arraySize );
16     cout << endl;
17     return 0; // indica terminação bem-sucedida
18 } // fim de main
19
20 // O que essa função faz?
21 void someFunction( int b[], int current, int size )
22 {
23     if ( current < size )
24     {
25         someFunction( b, current + 1, size );
26         cout << b[ current ] << " ";
27     } // fim do if
28 } // fim da função someFunction

```

7.22 Utilize um array bidimensional para resolver o seguinte problema. Uma empresa tem quatro equipes de vendas (1 a 4) que vendem cinco produtos diferentes (1 a 5). Uma vez por dia, cada vendedor passa uma nota de cada tipo de produto diferente vendido. Cada nota contém o seguinte:

- O número do vendedor
- O número do produto
- O valor total em reais desse produto vendido nesse dia

Portanto, cada vendedor passa entre 0 e 5 notas de vendas por dia. Suponha que as informações a partir de todas as notas durante o último mês estão disponíveis. Escreva um programa que lerá todas essas informações para as vendas do último mês e resumirá as vendas totais por vendedor e produto. Todos os totais devem ser armazenados no array bidimensional `sales`. Depois de processar todas as informações durante o último mês, imprima os resultados em formato tabular com cada uma das colunas representando um vendedor específico e cada uma das linhas representando um produto específico. Cruze cada linha de total para obter as vendas totais de cada produto durante o último mês; cruze cada coluna de total para obter as vendas totais por vendedor durante o último mês. Sua impressão tabular deve incluir esses totais cruzados à direita das linhas totalizadas e na parte inferior das colunas totalizadas.

7.23 (Gráficos de tartaruga) A linguagem Logo, que é popular entre crianças do ensino fundamental, tornou famoso o conceito de *gráficos de tartaruga*. Imagine uma tartaruga mecânica que caminha em um ambiente sob o controle de um programa C++. A tartaruga segura uma caneta em uma de duas posições, para cima ou para baixo. Enquanto a caneta está para baixo, a tartaruga desenha formas à medida que se move; enquanto a caneta está para cima, a tartaruga move-se quase livremente sem escrever nada. Nesse problema você simulará a operação da tartaruga e criará também um *sketchpad* computadorizado.

Utilize um array de 20 por 20 `floor` que é inicializado como zeros. Leia comandos a partir de um array que contenha esses comandos. Monitore a posição atual da tartaruga todas as vezes e se a caneta está atualmente para cima ou para baixo. Suponha que a tartaruga sempre inicia na posição (0, 0) do chão com sua caneta para cima. O conjunto de comandos da tartaruga que seu programa deve processar é mostrado na Figura 7.33.

Suponha que a tartaruga esteja em algum lugar próximo ao centro do chão. O seguinte ‘programa’ desenharia e imprimiria um quadrado de 12 por 12 e terminaria com a caneta na posição para cima:

```
2
5,12
3
5,12
3
5,12
3
5,12
1
6
9
```

À medida que a tartaruga se move com a caneta para baixo, configure os elementos apropriados do array `floor` como 1s. Quando o comando 6 (imprimir) é dado, onde quer que haja um 1 no array, exiba um asterisco ou algum caractere diferente que você escolher. Onde quer que haja um zero, exiba uma lacuna. Escreva um programa para implementar as capacidades dos gráficos de tartaruga discutidos aqui. Escreva vários programas de gráfico de tartaruga para desenhar formas interessantes. Adicione outros comandos para aumentar as capacidades de sua linguagem de gráfico de tartaruga.

7.24 (*O Passeio do Cavalo*) Um dos quebra-cabeças mais interessantes dos entusiastas do xadrez é o problema do Passeio do Cavalo. Esta é a pergunta: a peça de xadrez chamada cavalo pode mover-se em um tabuleiro vazio e tocar cada um dos 64 quadrados uma vez e unicamente uma vez? Estudamos esse problema intrigante profundamente nesse exercício.

O cavalo move-se em um caminho em forma de L (duas posições em uma direção e então uma em uma direção perpendicular). Portanto, a partir de um quadrado no meio de um tabuleiro vazio, o cavalo pode fazer oito movimentos diferentes (numerados de 0 a 7) como mostra a Figura 7.34.

- Desenhe um tabuleiro de 8 por 8 em uma folha de papel e experimente fazer o Passeio do Cavalo à mão. Coloque um 1 no primeiro quadrado para o qual você move o cavalo, um 2 no segundo quadrado, um 3 no terceiro etc. Antes de iniciar o passeio, estime aonde você imagina chegar, lembrando que um passeio completo consiste em 64 movimentos. Até onde você foi? Isso foi próximo de sua estimativa?
- Agora vamos desenvolver um programa que moverá o cavalo por um tabuleiro. O tabuleiro é representado por um array bidimensional 8 por 8 chamado `board`. Cada um dos quadrados é inicializado como zero. Descrevemos cada um dos oito possíveis movimentos em termos de seus componentes verticais e horizontais. Por exemplo, um movimento do tipo 0 como mostrado na Figura 7.34 consiste em mover dois quadrados horizontalmente para direita e um quadrado verticalmente para cima. O movimento 2 consiste em mover um quadrado horizontalmente para a esquerda e dois quadrados verticalmente para cima. Movimentos horizontais para a esquerda e movimentos verticais para cima são indicados com números negativos. Os oito movimentos podem ser descritos por dois arrays unidimensionais, `horizontal` e `vertical`, como segue:

```
horizontal[ 0 ] = 2
horizontal[ 1 ] = 1
horizontal[ 2 ] = -1
horizontal[ 3 ] = -2
horizontal[ 4 ] = -2
horizontal[ 5 ] = -1
horizontal[ 6 ] = 1
horizontal[ 7 ] = 2

vertical[ 0 ] = -1
vertical[ 1 ] = -2
vertical[ 2 ] = -2
vertical[ 3 ] = -1
vertical[ 4 ] = 1
vertical[ 5 ] = 2
vertical[ 6 ] = 2
vertical[ 7 ] = 1
```

Comando	Significado
1	Caneta para cima
2	Caneta para baixo
3	Vira para a direita
4	Vira para a esquerda
5,10	Mova para a frente 10 espaços (ou um número diferente de 10)
6	Imprima o array 20 por 20
9	Fim dos dados (sentinela)

Figura 7.33 Comandos dos gráficos de tartaruga.

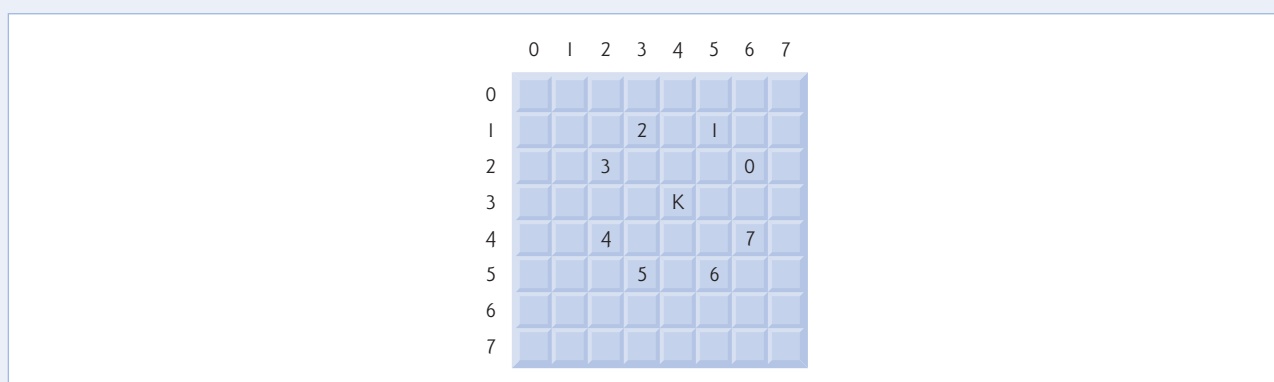


Figura 7.34 Os oito possíveis movimento do cavalo.

Faça com que as variáveis `currentRow` e `currentColumn` indiquem a linha e a coluna da posição atual do cavalo. Para fazer um movimento do tipo `moveNumber`, onde `moveNumber` está entre 0 e 7, seu programa utiliza as instruções

```
currentRow += vertical[ moveNumber ];
currentColumn += horizontal[ moveNumber ];
```

Mantenha um contador que varia de 1 para 64. Registre a última contagem em cada quadrado para o qual o cavalo se move. Lembre-se de testar cada potencial movimento para ver se o cavalo já visitou esse quadrado e, naturalmente, teste cada potencial movimento para certificar-se de que o cavalo não caia fora do tabuleiro. Agora escreva um programa para mover o cavalo pelo tabuleiro. Execute o programa. Quantos movimentos o cavalo fez?

- c) Depois de tentar escrever e executar um programa para o Passeio do Cavalo, provavelmente alguns insights valiosos foram desenvolvidos. Utilizaremos esses insights para desenvolver uma **heurística** (ou estratégia) para mover o cavalo. A heurística não garante sucesso, mas uma heurística cuidadosamente desenvolvida aprimora significativamente a chance de sucesso. Você pode ter observado que os quadrados externos são mais incômodos que os quadrados próximos do centro do tabuleiro. De fato, os quadrados mais problemáticos, ou inacessíveis, são os quatro cantos.

A intuição pode sugerir que você deve tentar mover o cavalo para os quadrados mais problemáticos primeiro e deixar abertos aqueles que são fáceis de alcançar de modo que, quando o tabuleiro ficar congestionado próximo do fim do passeio, haja maior chance de sucesso.

Podemos desenvolver uma 'heurística de acessibilidade' classificando cada quadrado de acordo com seu grau de acessibilidade e então sempre mover o cavaleiro para o quadrado (dentro do movimento em forma de L do cavalo, naturalmente) que seja mais inacessível. Rotulamos um array bidimensional `accessibility` com números indicando a partir de quantos quadrados cada quadrado particular é acessível. Em um tabuleiro vazio, os 16 quadrados centrais são avaliados como 8, cada canto é avaliado como 2 e os outros quadrados têm números de acessibilidade 3, 4 ou 6:

```

2 3 4 4 4 4 3 2
3 4 6 6 6 6 4 3
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
4 6 8 8 8 8 6 4
3 4 6 6 6 6 4 3
2 3 4 4 4 4 3 2

```

Agora escreva uma versão do programa do Passeio do Cavalo utilizando a heurística de acessibilidade. Em qualquer dado momento, o cavaleiro deve mover-se para o quadrado com o número mais baixo de acessibilidade. Em caso de um impasse, o cavalo pode mover-se para qualquer quadrado já visitado. Portanto, o passeio pode iniciar em qualquer um dos quatro cantos. [Nota: À medida que o cavalo se move pelo tabuleiro de xadrez, seu programa deve reduzir os números de acessibilidade porque cada vez mais quadrados se tornam ocupados. Dessa maneira, em qualquer dado tempo durante o passeio, o número de acessibilidade de cada quadrado disponível permanecerá precisamente igual ao número de quadrados a partir dos quais esse quadrado pode ser alcançado.] Execute essa versão de seu programa. Você obteve um passeio completo? Agora modifique o programa para executar 64 passeios, um iniciando a partir de cada quadrado do tabuleiro de xadrez. Quantos passeios completos você obteve?

- d) Escreva uma versão do programa Passeio do Cavalo que, diante de um impasse entre dois ou mais quadrados, decide qual quadrado escolher olhando para a frente aqueles quadrados alcançáveis a partir dos quadrados geradores do impasse. Seu programa deve mover-se para o quadrado por meio do qual seu próximo movimento chegaria ao quadrado com o número de acessibilidade mais baixo.

7.25 (*Passeio do Cavalo: abordagens de força bruta*) No Exercício 7.24 desenvolvemos uma solução para o problema do Passeio do Cavalo. A abordagem utilizada, chamada de ‘acessibilidade heurística’, gera muitas soluções e executa eficientemente.

À medida que os computadores continuam aumentando em potência, seremos capazes de resolver cada vez mais problemas com a pura capacidade do computador e algoritmos relativamente simples. Essa é a abordagem de ‘força bruta’ para resolução de problemas.

- Utilize geração de números aleatórios para permitir que o cavalo ande pelo tabuleiro de xadrez (em seus movimentos válidos em forma de L, naturalmente) de maneira aleatória. Seu programa deve executar um passeio e imprimir o tabuleiro de xadrez final. Até onde o cavalo chegou?
- Muito provavelmente, o programa precedente produziu um passeio relativamente curto. Agora modifique seu programa para tentar 1.000 passeios. Utilize um array unidimensional para monitorar o número de passeios de cada comprimento. Quando seu programa termina de tentar os 1.000 passeios, ele deve imprimir essas informações em formato tabular organizado. Qual foi o melhor resultado?
- Muito provavelmente, o programa precedente deu-lhe alguns passeios ‘respeitáveis’, mas nenhum passeio completo. Agora ‘solte todas as amarras’ e simplesmente deixe seu programa executar até produzir um passeio completo. [Atenção: Essa versão do programa pode executar durante horas em um computador poderoso.] Mais uma vez, mantenha uma tabela do número de passeios de cada comprimento e imprima essa tabela quando o primeiro passeio completo for encontrado. Quantos passeios seu programa tentou antes de produzir um passeio completo? Quanto tempo ele levou?
- Compare a versão de força bruta do Passeio do Cavalo com a versão de acessibilidade heurística. Qual exigiu um estudo mais cuidadoso do problema? Qual algoritmo foi mais difícil de desenvolver? Qual exigiu mais capacidade do computador? Poderíamos ter certeza (antecipadamente) de obter um passeio completo com a abordagem de acessibilidade heurística? Poderíamos ter certeza (com antecedência) de obter um passeio completo com a abordagem de força bruta? Argumente as vantagens e desvantagens de resolver problema de força bruta em geral.

7.26 (*Oito Rainhas*) Outro problema difícil para fãs do xadrez é o das Oito Rainhas. Eis o problema: É possível colocar oito rainhas em um tabuleiro de xadrez vazio de modo que nenhuma rainha esteja ‘atacando’ qualquer outra, isto é, sem que duas rainhas estejam na mesma linha, na mesma coluna ou na mesma diagonal? Utilize a consideração desenvolvida no Exercício 7.24 a fim de formular uma heurística para resolver o problema das Oito Rainhas. Execute seu programa. [Dica: É possível atribuir um valor para cada quadrado do tabuleiro de xadrez que indica quantos quadrados de um tabuleiro de xadrez vazio ‘são eliminados’ se uma rainha for colocada nesse quadrado. A cada um dos cantos seria atribuído o valor 22, como na Figura 7.35.] Uma vez que esses ‘números de eliminação’ são colocados em todos os 64 quadrados, uma heurística apropriada talvez seja: coloque a próxima rainha no quadrado com o menor número de eliminação. Por que essa estratégia é intuitivamente atraente?

7.27 (*Oito Rainhas: abordagens de força bruta*) Nesse exercício você desenvolverá várias abordagens de força bruta para resolver o problema das Oito Rainhas introduzido no Exercício 7.26.

- Resolva o exercício das Oito Rainhas utilizando a técnica de força bruta aleatória desenvolvida no Exercício 7.25.
- Utilize uma técnica exaustiva, isto é, tente todas as combinações possíveis de oito rainhas no tabuleiro de xadrez.
- Por que você supõe que a abordagem de força bruta exaustiva pode não ser apropriada para resolver o problema do Passeio do Cavalo?
- Compare e contraste as abordagens de força bruta aleatória e de força bruta exaustiva em geral.

7.28 (*Passeio do Cavalo: Teste do Passeio Fechado*) No Passeio do Cavalo, um passeio completo ocorre quando o cavalo move-se tocando cada um dos 64 quadrados do tabuleiro de xadrez uma vez e unicamente uma vez. Um passeio fechado ocorre quando o 64º movimento

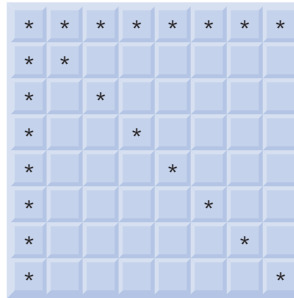


Figura 7.35 Os 22 quadrados foram eliminados posicionando uma rainha no canto esquerdo superior.

cai no quadrado em que o cavalo iniciou o passeio. Modifique o programa do Passeio do Cavalo que você escreveu no Exercício 7.24 a fim de testar para um passeio fechado se um passeio completo ocorreu.

7.29 (*O Crivo de Eratóstenes*) Um inteiro primo é qualquer inteiro que seja divisível apenas por si mesmo e por 1. O crivo, ou peneira, de Eratóstenes é um método para localizar números primos. Ele opera como segue:

- Crie um array com todos os elementos inicializados como 1 (verdadeiro). Os elementos do array com subscritos primos permanecerão como 1. Todos os outros elementos do array acabarão sendo configurados como zero. Você ignorará os elementos 0 e 1 nesse exercício.
- Iniciando com o subscrito de array 2 (o subscrito 1 deve ser primo), toda vez que for localizado um elemento do array cujo valor é 1, faça um loop pelo restante do array e configure como zero cada elemento cujo subscrito é um múltiplo do subscrito para o elemento com valor 1. Para o subscrito de array 2, todos os elementos além de 2 no array que são múltiplos de 2 serão configurados como zero (subscritos 4, 6, 8, 10 etc.); para o subscrito de array 3, todos os elementos além de 3 no array que são múltiplos de 3 serão configurados como zero (subscritos 6, 9, 12, 15 etc.); e assim por diante.

Quando esse processo estiver completo, os elementos do array que ainda estiverem configurados como 1 indicam que o subscrito é um número primo. Esses subscritos então podem ser impressos. Escreva um programa que utiliza um array de 1.000 elementos para determinar e imprimir os números primos entre 2 e 999. Ignore o elemento 0 do array.

7.30 (*Bucket Sort*) Uma classificação do tipo **bucket sort** inicia com um array unidimensional de inteiros positivos a ser classificado e um array bidimensional de inteiros com linhas indexadas de 0–9 e colunas indexadas de 0 a $n - 1$, onde n é o número dos valores no array a serem classificados. Cada linha do array bidimensional é chamada *bucket*. Escreva uma função `bucketSort` que aceita um array de inteiros e o tamanho do array como argumentos e funciona da seguinte maneira:

- Coloque cada valor do array unidimensional em uma linha do array de *bucket* com base no dígito das unidades do valor. Por exemplo, 97 é colocado na linha 7, 3 é colocado na linha 3, e 100 é colocado na linha 0. Isso é chamado de ‘passagem de distribuição’.
- Realize um loop pelo array de *bucket* linha por linha e copie os valores de volta para o array original. Isso é chamado uma ‘passagem de coleta’. A nova ordem dos valores precedentes no array unidimensional é 100, 3 e 97.
- Repita esse processo para a posição de cada dígito subsequente (dezenas, centenas, milhares etc.)

Na segunda passagem, 100 é colocado na linha 0, 3 é colocado na linha 0 (porque 3 não tem dígito de dezenas) e 97 é colocado na linha 9. Depois da passagem de coleta, a ordem dos valores no array unidimensional é 100, 3 e 97. Na terceira passagem, 100 é colocado na linha 1, 3 é colocado na linha zero, e 97 é colocado na linha zero (depois do 3). Depois da última passagem de coleta, o array original está agora na ordem classificada.

Observe que o array de *buckets* bidimensional tem 10 vezes o tamanho do array de inteiros sendo classificado. Essa técnica de classificação fornece melhor desempenho que uma classificação por inserção, mas exige muito mais memória. A classificação por inserção requer espaço para apenas um elemento de dados adicional. Esse é um exemplo da relação de troca espaço-tempo: A *bucket sort* utiliza mais memória que a classificação por inserção, mas seu desempenho é melhor. Essa versão da *bucket sort* requer cópia de todos os dados de volta para o array original a cada passagem. Outra possibilidade é criar um segundo array de *bucket* bidimensional e permutar os dados repetidamente entre os dois arrays de *bucket*.

Exercícios com recursão

7.31 (*Classificação por seleção*) A **classificação por seleção** pesquisa um array procurando o menor elemento. Então, o elemento menor é trocado com o primeiro elemento do array. O processo é repetido para o subarray que inicia com o segundo elemento do array. Cada passagem do array resulta em um elemento sendo colocado em sua localização adequada. O desempenho dessa classificação é comparável ao da classificação por inserção — para um array de n elementos, $n - 1$ passagens devem ser feitas e para cada subarray, $n - 1$ compara-

ções devem ser feitas para localizar o menor valor. Quando o subarray sendo processado contiver um elemento, o array será classificado. Escreva a função recursiva `selectionSort` para realizar esse algoritmo.

- 7.32** (*Palíndromos*) Um palíndromo é uma string que é lida da mesma maneira, quer da esquerda para a direita, quer da direita para a esquerda. Alguns exemplos de palíndromos são 'radar', 'Atlas salta' e (se espaços forem ignorados) 'erro comum ocorre'. Escreva uma função recursiva `testPalindrome` que retorna `true` se a string armazenada no array for um palíndromo e `false` caso contrário. A função deve ignorar espaços e pontuação na string.
- 7.33** (*Pesquisa linear*) Modifique o programa na Figura 7.19 de modo que ele utilize a função recursiva `linearSearch` a fim de realizar uma pesquisa linear do array. A função deve receber um array de inteiros e o tamanho do array como argumentos. Se a chave de pesquisa for encontrada, retorne o subscrito do array; caso contrário, retorne `-1`.
- 7.34** (*Oito Rainhas*) Modifique o programa das Oito Rainhas criado no Exercício 7.26 para resolver o problema recursivamente.
- 7.35** (*Imprimir um array*) Escreva uma função recursiva `printArray` que aceita um array, um subscrito inicial e um subscrito final como argumentos e nada retorna. A função deve parar o processamento e retornar quando o subscrito inicial for igual ao subscrito final.
- 7.36** (*Imprimir uma string de trás para frente*) Escreva uma função recursiva `stringReverse` que aceita um array de caracteres contendo uma string e um subscrito inicial como argumento, imprime a string de trás para frente e nada retorna. A função deve parar o processamento e retornar quando o caractere nulo de terminação for encontrado.
- 7.37** (*Localizar o valor mínimo em um array*) Escreva uma função recursiva `recursiveMinimum` que aceita um array de inteiros, um subscrito inicial e um subscrito final como argumentos e retorna o menor elemento do array. A função deve parar o processamento e retornar quando o subscrito inicial for igual ao subscrito final.

Exercícios com vector

- 7.38** Utilize um `vector` de inteiros para resolver o problema descrito no Exercício 7.10.
- 7.39** Modifique o programa de jogo de dados que você criou no Exercício 7.17 de modo que ele utilize um `vector` para armazenar os números de vezes que cada possível soma dos dois dados aparece.
- 7.40** (*Localizar o valor mínimo em um vector*) Modifique sua solução do Exercício 7.37 para localizar o valor mínimo em um `vector` em vez de em um array.



*Os endereços nos são dados
para ocultar nosso paradeiro.*
Saki (H. H. Munro)

*Usando de cautela e
circunlóquios, chegamos ao
caminho por desvios.*
William Shakespeare

*Muitas coisas, quando
tudo parece apontar para o
consenso, podem funcionar de
maneira adversa.*
William Shakespeare

*Você descobrirá que é uma
prática muito boa sempre
verificar suas referências,
senhor!*
Dr. Routh

Ponteiros e strings baseadas em ponteiro

OBJETIVOS

Neste capítulo, você aprenderá:

- O que são ponteiros.
- As semelhanças e diferenças entre ponteiros e referências e quando utilizar cada um.
- Como utilizar ponteiros para passar argumentos a funções por referência.
- Como utilizar strings no estilo C baseadas em ponteiro.
- Os estreitos relacionamentos entre ponteiros, arrays e strings no estilo C.
- Como utilizar ponteiros para funções.
- A declarar e utilizar arrays de strings no estilo C.

- 8.1 Introdução
- 8.2 Declarações de variável ponteiro e inicialização
- 8.3 Operadores de ponteiro
- 8.4 Passando argumentos para funções por referência com ponteiros
- 8.5 Utilizando const com ponteiros
- 8.6 Classificação por seleção utilizando passagem por referência
- 8.7 Operadores sizeof
- 8.8 Expressões e aritmética de ponteiro
- 8.9 Relacionamento entre ponteiros e arrays
- 8.10 Arrays de ponteiros
- 8.11 Estudo de caso: simulação de embaralhamento e distribuição de cartas
- 8.12 Ponteiros de função
- 8.13 Introdução ao processamento de string baseada em ponteiro
 - 8.13.1 Fundamentos de caracteres e strings baseadas em ponteiro
 - 8.13.2 Funções de manipulação de string da biblioteca de tratamento de strings
- 8.14 Síntese

Resumo | Terminologia | Exercícios de revisão | Respostas dos exercícios de revisão | Exercícios |
 Seção Especial: construindo seu próprio computador | Mais exercícios sobre ponteiros |
 Exercícios de manipulação de string | Seção especial: exercícios avançados de manipulação de string |
 Um projeto desafiador de manipulação de string

8.1 Introdução

Este capítulo discute um dos recursos mais poderosos da linguagem de programação C++, o ponteiro. No Capítulo 6, vimos que as referências podem ser utilizadas para realizar a passagem por referência. Os ponteiros também permitem a passagem por referência e podem ser utilizados para criar e manipular estruturas de dados dinâmicas (isto é, estruturas de dados que podem crescer e encolher), como listas vinculadas, filas, pilhas e árvores. Este capítulo explica conceitos básicos do ponteiro e reforça o relacionamento íntimo entre arrays e ponteiros. A visão de arrays como ponteiros deriva da linguagem de programação C. Como vimos no Capítulo 7, a classe C++ Standard Library `vector` fornece uma implementação de arrays como objetos completos.

De modo semelhante, o C++ realmente oferece dois tipos de strings — objetos da classe `string` (que utilizamos desde o Capítulo 3) e strings `char *` baseadas em ponteiro no estilo C. Este capítulo sobre ponteiros discute as strings `char *` para aprofundar seu conhecimento de ponteiros. De fato, as strings terminadas por caractere nulo introduzidas na Seção 7.4 e utilizadas na Figura 7.12 são strings `char *` baseadas em ponteiro. Este capítulo também inclui uma coleção considerável de exercícios de processamento de strings que utilizam strings `char *`. As strings `char *` baseadas em ponteiro no estilo C são amplamente utilizadas em sistemas C e C++ legados. Então, se trabalha com sistemas C ou C++ legados, você pode ser solicitado a manipular essas strings `char *` baseadas em ponteiro.

Examinaremos o uso de ponteiros com classes no Capítulo 13, “Programação orientada a objetos: polimorfismo”, em que veremos que o chamado ‘processamento polimórfico’ de programação orientada a objetos é realizado com ponteiros e referências. O Capítulo 21, “Estruturas de dados”, apresenta exemplos de como criar e utilizar estruturas de dados dinâmicas que são implementadas com ponteiros.

8.2 Declarações de variável ponteiro e inicialização

As variáveis ponteiro contêm endereços de memória como seus valores. Normalmente, uma variável contém diretamente um valor específico. Mas um ponteiro contém o endereço de memória de uma variável que, por sua vez, contém um valor específico. Nesse sentido, o nome de uma variável **referencia um valor diretamente** e um ponteiro **referencia um valor indiretamente** (Figura 8.1). Referenciar um valor por um ponteiro é freqüentemente chamado de **indireção**. Observe que os diagramas normalmente representam um ponteiro como uma seta da variável que contém um endereço para a variável localizada nesse endereço na memória.

Os ponteiros, como qualquer outra variável, devem ser declarados antes de ser utilizados. Por exemplo, para o ponteiro na Figura 8.1, a declaração

```
int *countPtr, count;
```

declara a variável `countPtr` como do tipo `int *` (isto é, um ponteiro para um valor `int`) e exibe ‘`countPtr` é um ponteiro para `int`’ ou ‘`countPtr` aponta para um objeto tipo `int`’. Além disso, a variável `count` na declaração anterior é declarada como um `int`, não como

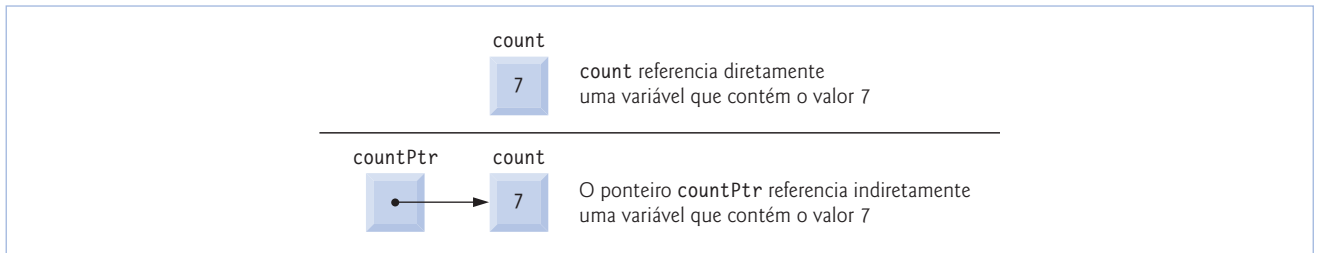


Figura 8.1 Referenciando direta e indiretamente uma variável.

um ponteiro para um `int`. O `*` na declaração se aplica apenas a `countPtr`. Toda variável sendo declarada como um ponteiro deve ser precedida por um asterisco (`*`). Por exemplo, a declaração

```
double *xPtr, *yPtr;
```

indica que tanto `xPtr` como `yPtr` são ponteiros para valores `double`. Quando `*` aparece em uma declaração, ele não é um operador; em vez disso, indica que a variável sendo declarada é um ponteiro. Os ponteiros podem ser declarados para apontar para objetos de qualquer tipo de dados.



Erro comum de programação 8.1

Supor que o `*` utilizado para declarar um ponteiro distribui-se por todos os nomes de variável na lista de variáveis separadas por vírgulas de uma declaração pode levar a erros. Cada ponteiro deve ser declarado com o `*` prefixado ao nome (com ou sem espaço entre eles — o compilador ignora o espaço). Declarar apenas uma variável por declaração ajuda a evitar esse tipo de erro e melhora a legibilidade do programa.



Boa prática de programação 8.1

Embora não seja um requisito, incluir as letras `Ptr` nos nomes de variáveis ponteiro torna claro que essas variáveis são ponteiros e devem ser tratadas apropriadamente.

Os ponteiros devem ser inicializados quando forem declarados ou em uma atribuição. Um ponteiro pode ser inicializado como `0`, `NULL` ou como um endereço. Um ponteiro com o valor `0` ou `NULL` não aponta para nada e é conhecido como um **ponteiro nulo**. A constante simbólica `NULL` é definida no arquivo de cabeçalho `<iostream>` (e em vários outros arquivos de cabeçalho de biblioteca-padrão) para representar o valor `0`. Inicializar um ponteiro como `NULL` é equivalente a inicializar um ponteiro como `0`, mas em C++, o `0` é utilizado por convenção. Quando `0` é atribuído, ele é convertido em um ponteiro do tipo apropriado. O valor `0` é o único valor do tipo inteiro que pode ser atribuído diretamente a uma variável ponteiro sem primeiro fazer coerção do inteiro em um tipo ponteiro. A atribuição de endereço numérico de uma variável a um ponteiro é discutida na Seção 8.3.



Dica de prevenção de erro 8.1

Inicialize ponteiros para impedir apontar para áreas da memória desconhecidas ou não inicializadas.

8.3 Operadores de ponteiro

O **operador de endereço** (`&`) é um operador unário que retorna o endereço de memória de seu operando. Por exemplo, considerando as declarações

```
int y = 5; // declara variável y
int *yPtr; // declara variável ponteiro yPtr
```

a instrução

```
yPtr = &y; // atribui o endereço de y a yPtr
```

atribui o endereço da variável `y` à variável ponteiro `yPtr`. Então dizemos que a variável `yPtr` ‘aponta para’ `y`. Agora, `yPtr` referencia indiretamente o valor da variável `y`. Observe que o uso do `&` na instrução de atribuição anterior não é o mesmo uso do `&` em uma declaração de variável de referência, que sempre é precedida por um nome de tipo de dados.

A Figura 8.2 mostra uma representação esquemática da memória depois da atribuição precedente. O ‘relacionamento de apontar’ é indicado desenhando uma seta a partir da caixa que representa o ponteiro `yPtr` na memória até a caixa que representa a variável `y` na memória.

A Figura 8.3 mostra outra representação do ponteiro na memória, supondo que a variável do tipo inteiro `y` está armazenada na posição da memória `600000` e que a variável ponteiro `yPtr` é armazenada na posição da memória `500000`. O operando do operador de

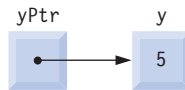


Figura 8.2 Representação gráfica de um ponteiro que aponta para uma variável na memória.



Figura 8.3 Representação de y e yPtr na memória.

endereço deve ser um *lvalue* (isto é, algo a que um valor pode ser atribuído, como um nome de variável ou uma referência); o operador de endereço não pode ser aplicado a constantes ou expressões que não resultam em referências.

O **operador ***, comumente referido como **operador de indireção** ou **operador de desreferência**, retorna um sinônimo (isto é, um alias ou um apelido) ao objeto para o qual seu operando de ponteiro aponta. Por exemplo (referindo-se novamente à Figura 8.2), a instrução

```
cout << *yPtr << endl;
```

imprime o valor de variável y, isto é, 5, assim como faria a instrução

```
cout << y << endl;
```

Utilizar o * dessa maneira é chamado de **desreferenciar um ponteiro**. Observe que um ponteiro desreferenciado também pode ser utilizado no lado esquerdo de uma instrução de atribuição, como em

```
*yPtr = 9;
```

que atribuiria 9 a y na Figura 8.3. O ponteiro desreferenciado também pode ser utilizado para receber um valor de entrada como em

```
cin >> *yPtr;
```

o que coloca o valor de entrada em y. O ponteiro desreferenciado é um *lvalue*.



Erro comum de programação 8.2

Desreferenciar um ponteiro que não foi inicializado de modo adequado ou que não foi atribuído para apontar a uma posição específica na memória poderia produzir um erro fatal em tempo de execução, ou poderia modificar acidentalmente dados importantes e permitir que o programa executasse até a conclusão, possivelmente com resultados incorretos.



Erro comum de programação 8.3

Uma tentativa de desreferenciar uma variável que não é um ponteiro é um erro de compilação.



Erro comum de programação 8.4

Desreferenciar um ponteiro nulo é normalmente um erro fatal em tempo de execução.

O programa na Figura 8.4 demonstra os operadores de ponteiro & e *. Neste exemplo, as posições da memória são enviadas para a saída por << como inteiros hexadecimais (isto é, base 16). (Consulte o Apêndice D, “Sistemas de numeração”, para obter informações adicionais sobre inteiros hexadecimais.) Observe que os endereços de memória hexadecimal enviados para a saída por esse programa dependem do compilador e do sistema operacional; portanto, você pode obter diferentes resultados ao executar o programa.



Dica de portabilidade 8.1

O formato em que um ponteiro é enviado para a saída depende do compilador. Alguns sistemas geram saída de valores de ponteiro como inteiros hexadecimais, enquanto outros sistemas utilizam inteiros decimais.

Note que o endereço de a (linha 15) e o valor de aPtr (linha 16) são idênticos na saída, confirmando que o endereço de a é, de fato, atribuído à variável ponteiro aPtr. Os operadores & e * são o oposto um do outro — quando os dois são aplicados consecutivamente a aPtr em qualquer ordem, eles se ‘cancelam’ e o mesmo resultado (o valor em aPtr) é impresso.

```

1 // Figura 8.4: fig08_04.cpp
2 // Utilizando os operadores & e *.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     int a; // a é um inteiro
10    int *aPtr; // aPtr é um ponteiro int * -- para um inteiro
11
12    a = 7; // atribuiu 7 a a
13    aPtr = &a; // atribui o endereço de a ao aPtr
14
15    cout << "The address of a is " << &a
16         << "\nThe value of aPtr is " << aPtr;
17    cout << "\n\nThe value of a is " << a
18         << "\nThe value of *aPtr is " << *aPtr;
19    cout << "\n\nShowing that * and & are inverses of "
20         << "each other.\n&*aPtr = " << &*aPtr
21         << "\n*aPtr = " << *aPtr << endl;
22    return 0; // indica terminação bem-sucedida
23 } // fim de main

```

```

The address of a is 0012F580
The value of aPtr is 0012F580

```

```

The value of a is 7
The value of *aPtr is 7

```

```

Showing that * and & are inverses of each other.
&*aPtr = 0012F580
*aPtr = 0012F580

```

Figura 8.4 Operadores de ponteiro & e *.

A Figura 8.5 lista a precedência e a associatividade dos operadores introduzidos até agora. Observe que o operador de endereço (&) e o de desreferência (*) são operadores unários no terceiro nível de precedência no gráfico.

8.4 Passando argumentos para funções por referência com ponteiros

Há três maneiras em C++ de passar argumentos para uma função — passagem por valor, **passagem por referência com argumentos de referência** e **passagem por referência com argumentos de ponteiro**. O Capítulo 6 comparou e contrastou a passagem por valor e a passagem por referência com argumentos de referência. Nesta seção, explicamos como passar por referência com argumentos de ponteiro.

Como vimos no Capítulo 6, `return` pode ser utilizado para retornar um valor de uma função chamada para um chamador (ou retornar o controle de uma função chamada sem passar um valor de volta). Vimos também que os argumentos podem ser passados para uma função usando argumentos de referência. Esses argumentos permitem que a função chamada modifique os valores originais dos argumentos no chamador. Os argumentos de referência também permitem aos programas passar grandes objetos de dados para uma função e evitam o overhead de passar os objetos por valor (o que, naturalmente, exige a produção de uma cópia do objeto). Os ponteiros, como as referências, também podem ser utilizados para modificar uma ou mais variáveis no chamador ou passar ponteiros para objetos grandes de dados a fim de evitar o overhead de passar os objetos por valor.

Em C++, os programadores podem utilizar ponteiros e o operador de indireção (*) para realizar a passagem por referência (exatamente como a passagem por referência é feita em programas C, porque o C não tem referências). Ao chamar uma função com um argumento que deve ser modificado, o endereço do argumento é passado. Isso normalmente é realizado aplicando o operador de endereço (&) ao nome da variável cujo valor será modificado.

Operadores	Associatividade	Tipo
() []	da esquerda para a direita	mais alto
++ -- <code>static_cast< tipo >(operando)</code>	da esquerda para a direita	unário (pós-fixado)
++ -- + - ! & *	da direita para a esquerda	unário (prefixo)
* / %	da esquerda para a direita	multiplicativo
+ -	da esquerda para a direita	aditivo
<< >>	da esquerda para a direita	inserção/extração
< <= > >=	da esquerda para a direita	relacional
== !=	da esquerda para a direita	igualdade
&&	da esquerda para a direita	E lógico
	da esquerda para a direita	OU lógico
?:	da direita para a esquerda	ternário condicional
= += -= *= /= %=	da direita para a esquerda	atribuição
,	da esquerda para a direita	vírgula

Figura 8.5 Precedência e associatividade de operadores.

Como vimos no Capítulo 7, os arrays não são passados com operador `&`, porque o nome do array é a posição inicial na memória do array (isto é, um nome de array já é um ponteiro). O nome de um array, `arrayName`, é equivalente a `&arrayName[ 0 ]`. Quando o endereço de uma variável é passado para uma função, o operador de indireção (`*`) pode ser utilizado na função para formar um sinônimo para o nome da variável — isso por sua vez pode ser utilizado para modificar o valor da variável naquela localização na memória do chamador.

As figuras 8.6 e 8.7 possuem duas versões presentes de uma função que eleva um inteiro ao cubo — `cubeByValue` e `cubeByReference`. A Figura 8.6 passa a variável `number` por valor para a função `cubeByValue` (linha 15). A função `cubeByValue` (linhas 21–24) eleva seu argumento ao cubo e passa o novo valor de volta para `main` utilizando uma instrução `return` (linha 23). O novo valor é atribuído a `number` (linha 15) em `main`. Observe que a função chamadora tem a oportunidade de examinar o resultado da chamada de função antes de modificar o valor da variável `number`. Por exemplo, nesse programa, poderíamos ter armazenado o resultado de `cubeByValue` em outra variável, examinado seu valor e atribuído o resultado a `number` somente depois de determinar que o valor retornado era razoável.

A Figura 8.7 passa a variável `number` para a função `cubeByReference` utilizando a passagem por referência com um argumento de ponteiro (linha 15) — o endereço de `number` é passado para a função. A função `cubeByReference` (linhas 22–25) especifica o parâmetro `nPtr` (um ponteiro para `int`) para receber seu argumento. A função desreferencia o ponteiro e eleva ao cubo o valor para o qual `nPtr` aponta (linha 24). Isso altera diretamente o valor de `number` em `main`.

```

1 // Figura 8.6: fig08_06.cpp
2 // Eleva uma variável ao cubo utilizando passagem por valor.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int cubeByValue( int ); // protótipo
8
9 int main()
10 {
11     int number = 5;
12
```

Figura 8.6 Passagem por valor utilizada para elevar o valor de uma variável ao cubo.

(continua)


```

13     cout << "The original value of number is " << number;
14
15     number = cubeByValue( number ); // passa number por valor ao cubeByValue
16     cout << "\nThe new value of number is " << number << endl;
17     return 0; // indica terminação bem-sucedida
18 } // fim de main
19
20 // calcula e retorna o cubo do argumento inteiro
21 int cubeByValue( int n )
22 {
23     return n * n * n; // eleva a variável local n ao cubo e retorna o resultado
24 } // fim da função cubeByValue

```

The original value of number is 5
The new value of number is 125

Figura 8.6 Passagem por valor utilizada para elevar o valor de uma variável ao cubo.

(continuação)

```

1 // Figura 8.7: fig08_07.cpp
2 // Eleva uma variável ao cubo usando passagem por referência com um argumento nPtr.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 void cubeByReference( int * ); // protótipo
8
9 int main()
10 {
11     int number = 5;
12
13     cout << "The original value of number is " << number;
14
15     cubeByReference( &number ); // passa endereço de number para cubeByReference
16
17     cout << "\nThe new value of number is " << number << endl;
18     return 0; // indica terminação bem-sucedida
19 } // fim de main
20
21 // calcula o cubo de *nPtr; modifica a variável number em main
22 void cubeByReference( int *nPtr )
23 {
24     *nPtr = *nPtr * *nPtr * *nPtr; // eleva *nPtr ao cubo
25 } // fim da função cubeByReference

```

The original value of number is 5
The new value of number is 125

Figura 8.7 Passagem por referência com um argumento de ponteiro utilizado para elevar ao cubo o valor de uma variável.



Erro comum de programação 8.5

Não desreferenciar um ponteiro quando é necessário fazer isso para obter o valor para o qual o ponteiro aponta é um erro.

Uma função que recebe um endereço como um argumento deve definir um parâmetro de ponteiro para receber o endereço. Por exemplo, o cabeçalho para função `cubeByReference` (linha 22) especifica que `cubeByReference` recebe o endereço de uma variável `int` (isto é, um ponteiro para um `int`) como um argumento, armazena o endereço localmente em `nPtr` e não retorna um valor.

O protótipo de função para `cubeByReference` (linha 7) contém `int *` entre parênteses. Como com outros tipos de variável, não é necessário incluir nomes de parâmetros de ponteiro em protótipos de função. Os nomes de parâmetro incluídos para propósitos de documentação são ignorados pelo compilador.

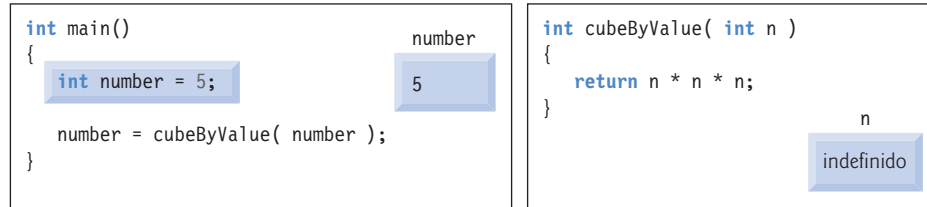
As figuras 8.8 e 8.9 analisam graficamente a execução dos programas das figuras 8.6 e 8.7, respectivamente.



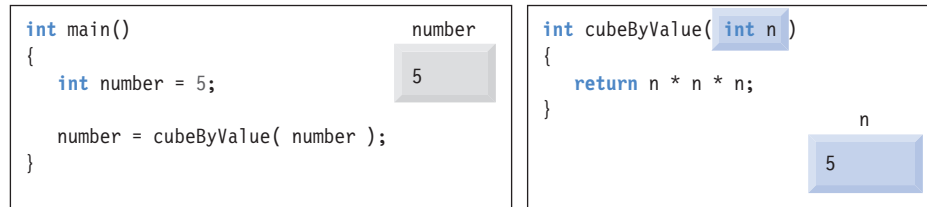
Observação de engenharia de software 8.1

Utilize a passagem por valor para passar argumentos para uma função a menos que o chamador requeira explicitamente que a função chamada modifique diretamente o valor da variável do argumento no chamador. Esse é outro exemplo do princípio do menor privilégio.

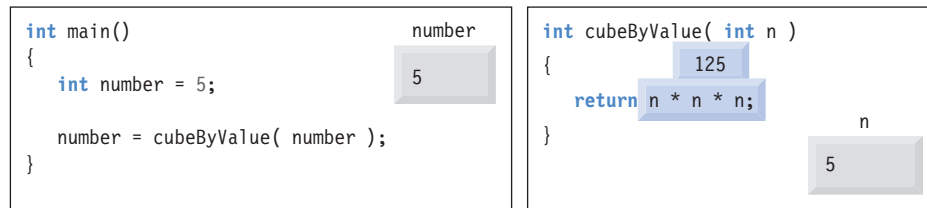
Passo 1: Antes de main chamar `cubeByValue`:



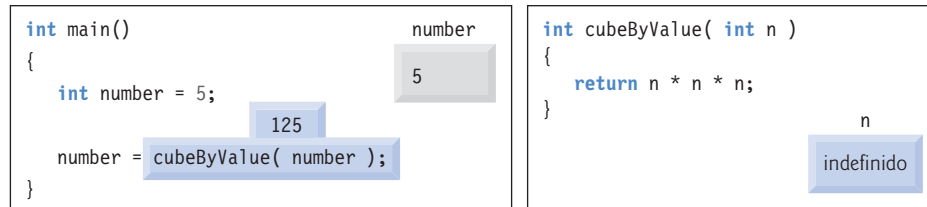
Passo 2: Depois de `cubeByValue` receber a chamada:



Passo 3: Depois de `cubeByValue` elevar o parâmetro `n` ao cubo e antes de `cubeByValue` retornar a main:



Passo 4: Depois de `cubeByValue` retornar a main e antes de atribuir o resultado a `number`:



Passo 5: Depois de main completar a atribuição a `number`:

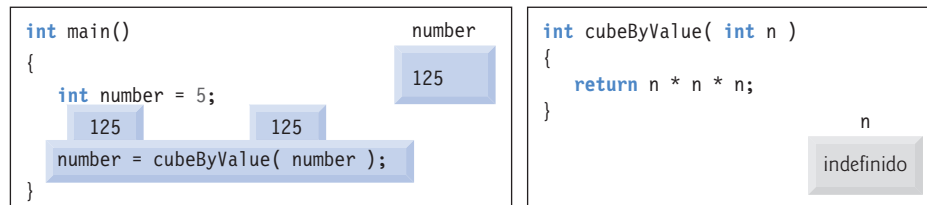
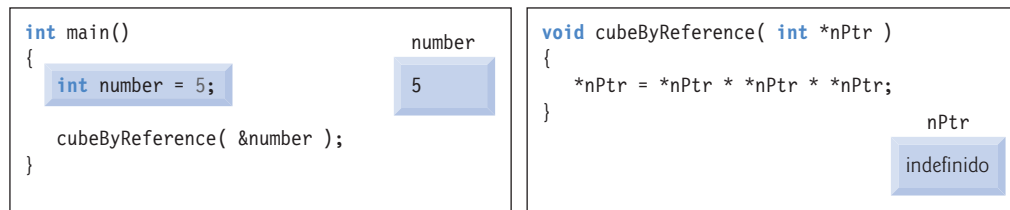
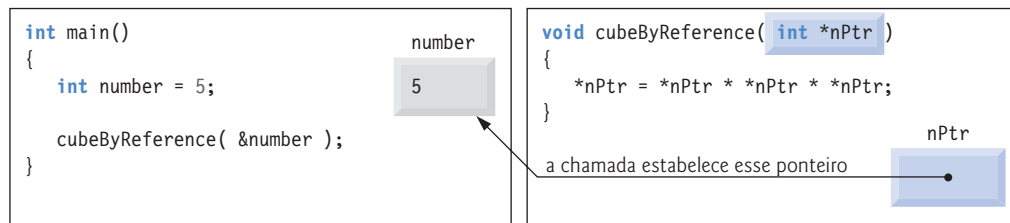


Figura 8.8 Análise da passagem por valor do programa da Figura 8.6.

Passo 1: Antes de main chamar cubeByReference:



Passo 2: Depois de cubeByReference receber a chamada e antes de *nPtr ser elevado ao cubo:



Passo 3: Depois de *nPtr ser elevado ao cubo e antes de o controle do programa retornar a main:

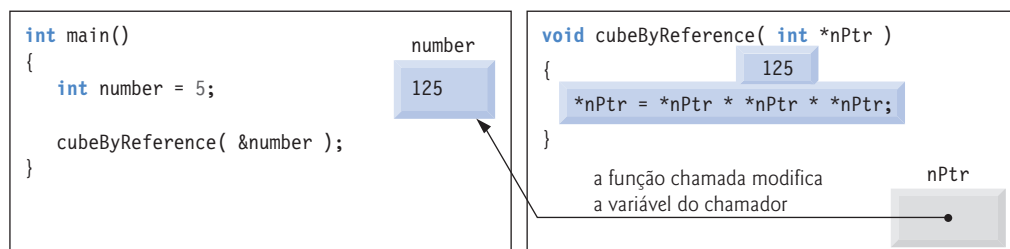


Figura 8.9 Análise da passagem por referência (com um argumento de ponteiro) do programa da Figura 8.7.

No cabeçalho e no protótipo de uma função que espera um array unidimensional como um argumento, pode-se utilizar a notação de ponteiro na lista de parâmetros de `cubeByReference`. O compilador não diferencia entre uma função que recebe um ponteiro e uma que recebe um array unidimensional. Isso, naturalmente, quer dizer que a função deve ‘saber’ quando está recebendo um array ou simplesmente uma única variável pela qual realizar a passagem por referência. Quando o compilador encontra um parâmetro de função para um array unidimensional na forma `int b[]`, o compilador converte o parâmetro em notação de ponteiro `int *b` (pronuncia-se ‘b é um ponteiro para um inteiro’). Ambas as formas de declarar um parâmetro de função como um array dimensional são intercambiáveis.

8.5 Utilizando const com ponteiros

Lembre-se de que o qualificador `const` permite ao programador informar ao compilador que o valor de uma variável particular não deve ser modificado.



Dica de portabilidade 8.2

Embora `const` seja bem definido em ANSI C e C++, alguns compiladores não o impõem adequadamente. Portanto, uma boa regra é: ‘Conheça seu compilador’.

Ao longo dos anos, nas primeiras versões do C, foi escrita uma grande base de código legado que não utilizava `const`, porque este qualificador não estava disponível. Por essa razão, há excelentes oportunidades de aprimorar a engenharia de software de código C antigo (também chamado de ‘legado’). Além disso, muitos programadores atualmente utilizando ANSI C e C++ não utilizam `const` em seus programas, porque começaram a programar nas primeiras versões de C. Esses programadores estão perdendo muitas oportunidades de aplicar boa engenharia de software.

Há muitas possibilidades para utilizar (ou não utilizar) `const` com parâmetros de função. Como escolher a mais apropriada dessas possibilidades? Deixe o princípio do menor privilégio ser seu guia. Sempre dê a uma função acesso suficiente aos dados em seus parâmetros para realizar sua tarefa especificada, porém não mais. Esta seção discute como combinar `const` com declarações de ponteiro para impor o princípio do menor privilégio.

O Capítulo 6 explicou que, quando uma função é chamada utilizando passagem por valor, uma cópia do argumento (ou argumentos) na chamada de função é feita e passada à função. Se a cópia é modificada na função, o valor original é mantido no chamador sem alteração.

Em muitos casos, um valor passado a uma função é modificado para que a função possa realizar sua tarefa. Entretanto, em alguns casos, o valor não deve ser alterado na função chamada, mesmo que a função chamada manipule apenas uma cópia do valor original.

Por exemplo, considere uma função que aceita um array unidimensional e seu tamanho como argumentos e, subsequentemente, imprime o array. Tal função deve fazer loop pelo array e gerar saída de cada elemento de array individualmente. O tamanho do array é utilizado no corpo da função para determinar o subscrito mais alto do array a fim de que o loop possa terminar quando a impressão for concluída. O tamanho do array não muda no corpo de função, então ele deve ser declarado como `const`. Naturalmente, como o array está apenas sendo impresso, ele também deve ser declarado como `const`. Isso é especialmente importante porque um array inteiro é *sempre* passado por referência e poderia ser facilmente alterado na função chamada.



Observação de engenharia de software 8.2

Se um valor não muda (ou não deve mudar) no corpo de uma função para o qual ele é passado, o parâmetro deve ser declarado como `const` para assegurar que ele não seja modificado acidentalmente.

Se houver uma tentativa de modificar um valor `const`, um aviso ou um erro é emitido, dependendo do compilador particular.



Dica de prevenção de erro 8.2

Antes de utilizar uma função, verifique seu protótipo de função para determinar os parâmetros que ele pode modificar.

Há quatro maneiras de passar um ponteiro para uma função: um ponteiro não constante para dados não constantes (Figura 8.10), um ponteiro não constante para dados constantes (figuras 8.11 e 8.12), um ponteiro constante para dados não constantes (Figura 8.13) e um ponteiro constante para dados constantes (Figura 8.14). Cada combinação fornece um nível diferente de privilégios de acesso.

Ponteiro não constante para dados não constantes

O acesso mais alto é concedido por um **ponteiro não constante para dados não constantes** — os dados podem ser modificados pelo ponteiro desreferenciado e o ponteiro pode ser modificado para apontar para outros dados. A declaração de um ponteiro não constante para dados não constantes não inclui `const`. Esse ponteiro pode ser utilizado para receber uma string terminada por caractere nulo em uma função que altera o valor de ponteiro para processar (e, possivelmente, modificar) cada caractere na string. A partir da Seção 7.4, lembre-se de que uma string terminada com caractere nulo pode ser colocada em um array de caracteres que contém os caracteres da string e um caractere nulo que indica onde a string termina.

```

1 // Figura 8.10: fig08_10.cpp
2 // Convertendo minúsculas em maiúsculas
3 // utilizando um ponteiro não constante para dados não constantes.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include <cctype> // protótipos para islower e toupper
9 using std::islower;
10 using std::toupper;
11
12 void convertToUpper( char * );
13
14 int main()
15 {
16     char phrase[] = "characters and $32.98";
17
18     cout << "The phrase before conversion is: " << phrase;
19     convertToUpper( phrase );
20     cout << "\nThe phrase after conversion is: " << phrase << endl;
21     return 0; // indica terminação bem-sucedida
22 } // fim de main
23
24 // converte string em letras maiúsculas
25 void convertToUpper( char *sPtr )

```

Figura 8.10 Convertendo uma string em letras maiúsculas.

(continua)

```

26 {
27     while ( *sPtr != '\0' ) // faz loop enquanto caractere atual não é '\0'
28     {
29         if ( islower( *sPtr ) ) // se o caractere estiver em minúsculas,
30             *sPtr = toupper( *sPtr ); // converte em maiúsculas
31
32         sPtr++; // move sPtr para o próximo caractere na string
33     } // fim do while
34 } // fim da função convertToUpper

```

The phrase before conversion is: characters and \$32.98
 The phrase after conversion is: CHARACTERS AND \$32.98

Figura 8.10 Convertendo uma string em letras maiúsculas.

(continuação)

Na Figura 8.10, a função `convertToUpper` (linhas 25–34) declara o parâmetro `sPtr` (linha 25) como sendo um ponteiro não constante para dados não constantes (novamente, `const` não é utilizado). A função processa um caractere por vez a partir da string terminada por caractere nulo armazenada no array de caracteres `phrase` (linhas 27–33). Tenha em mente que o nome de um array de caracteres na realidade equivale a um ponteiro para o primeiro caractere do array, então é possível passar `phrase` como um argumento para `convertToUpper`. A função `islower` (linha 29) aceita um argumento de caractere e retorna verdadeiro se o caractere for uma letra minúscula e falso, caso contrário. Os caracteres no intervalo de 'a' a 'z' são convertidos em suas letras maiúsculas correspondentes pela função `toupper` (linha 30); outros permanecem inalterados — a função `toupper` aceita um caractere como um argumento. Se o caractere for uma letra minúscula, a letra maiúscula correspondente será retornada; caso contrário, o caractere original será retornado. A função `toupper` e a `islower` fazem parte da biblioteca de tratamento de caracteres `<cctype>` (ver Capítulo 22, “Bits, caracteres, strings C e structs”). Depois de processar um caractere, a linha 32 incrementa `sPtr` por 1 (isso não seria possível se `sPtr` fosse declarado como `const`). Quando o operador `++` é aplicado a um ponteiro que aponta para um array, o endereço de memória armazenado no ponteiro é modificado para apontar para o próximo elemento do array (nesse caso, o próximo caractere na string). Adicionar um a um ponteiro é uma operação válida na **aritmética de ponteiros**, que é discutida em detalhes nas seções 8.8 e 8.9.

Ponteiro não constante para dados constantes

Um **ponteiro não constante para dados constantes** é um ponteiro que pode ser modificado para apontar para qualquer item de dados do tipo apropriado, mas os dados para os quais ele aponta não podem ser modificados por esse ponteiro. Esse ponteiro poderia ser utilizado para receber um argumento de array para uma função que irá processar cada elemento do array, mas não deve ter permissão de modificar os dados. Por exemplo, a função `printCharacters` (linhas 22–26 da Figura 8.11) declara o parâmetro `sPtr` (linha 22) para ser do tipo `const char *`, para que ele possa receber uma string baseada em ponteiro terminada por caractere nulo. A declaração é lida da direita para a esquerda como ‘`sPtr` é um ponteiro para uma constante do tipo caractere’. O corpo da função utiliza uma instrução `for` (linhas 24–25) para gerar saída de cada caractere na string até que o caractere nulo seja encontrado. Depois que cada caractere é impresso, o ponteiro `sPtr` é incrementado para apontar para o próximo caractere na string (isso funciona porque o ponteiro não é `const`). A função `main` cria o array `char phrase` para ser passado para `printCharacters`. Novamente, podemos passar o array `phrase` para `printCharacters` porque o nome do array é, na realidade, um ponteiro para o primeiro caractere no array.

```

1 // Figura 8.11: fig08_11.cpp
2 // Imprimindo uma string um caractere por vez utilizando
3 // um ponteiro não constante para dados constantes.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 void printCharacters( const char * ); // imprime usando ponteiro para dados const
9
10 int main()
11 {
12     const char phrase[] = "print characters of a string";

```

Figura 8.11 Imprimindo uma string um caractere por vez utilizando um ponteiro não constante para dados constantes.

(continua)

```

13
14     cout << "The string is:\n";
15     printCharacters( phrase ); // imprime caracteres em phrase
16     cout << endl;
17     return 0; // indica terminação bem-sucedida
18 } // fim de main
19
20 // sPtr pode ser modificado, mas não pode modificar o caractere para o qual
21 // ele aponta, isto é, sPtr é um ponteiro 'de leitura'
22 void printCharacters( const char *sPtr )
23 {
24     for ( ; *sPtr != '\0'; sPtr++ ) // nenhuma inicialização
25         cout << *sPtr; // exibe caractere sem modificação
26 } // fim da função printCharacters

```

```

The string is:
print characters of a string

```

Figura 8.11 Imprimindo uma string um caractere por vez utilizando um ponteiro não constante para dados constantes. (continuação)

A Figura 8.12 demonstra as mensagens de erro de compilação produzidas ao tentar compilar uma função que recebe um ponteiro não-constante para dados constantes e, então, tenta utilizar esse ponteiro para modificar os dados. [Nota: Lembre-se de que as mensagens de erro de compilador variam entre compiladores.]

Como sabemos, os arrays são tipos de dados agregados que armazenam itens de dados relacionados do mesmo tipo sob um nome. Quando uma função é chamada com um array como um argumento, o array é passado à função por referência. Entretanto, os objetos são sempre passados por valor — uma cópia do objeto inteiro é passada. Isso requer o overhead em tempo de execução de produzir uma cópia de cada item de dados no objeto e armazená-la na pilha de chamada de função. Quando um objeto deve ser passado para uma função, podemos utilizar um ponteiro para dados constantes (ou uma referência para dados constantes) para obter o desempenho da passagem por referência e a proteção da passagem por valor. Quando um ponteiro for passado para um objeto, deve-se fazer apenas uma cópia do endereço do objeto; o objeto em si não é copiado. Em uma máquina com endereços de quatro bytes, uma cópia de quatro bytes de memória é feita em vez de uma cópia de um objeto possivelmente grande.

```

1 // Figura 8.12: fig08_12.cpp
2 // Tentando modificar dados por meio de um
3 // ponteiro não-constante para dados constantes.
4
5 void f( const int * ); // protótipo
6
7 int main()
8 {
9     int y;
10
11     f( &y ); // f tenta modificação não-válida
12     return 0; // indica terminação bem-sucedida
13 } // fim de main
14
15 // xPtr não pode modificar o valor da variável constante para a qual ele aponta
16 void f( const int *xPtr )
17 {
18     *xPtr = 100; // erro: não é possível modificar objeto const
19 } // fim da função f

```

Figura 8.12 Tentando modificar dados por meio de um ponteiro não-constante para dados constantes. (continua)

Mensagem de erro do compilador de linha de comando Borland C++:

```
Error E2024 fig08_12.cpp 18:
    Cannot modify a const object in function f(const int *)
```

Mensagem de erro do compilador Microsoft Visual C++:

```
c:\cpphttp5_examples\ch08\Fig08_12\fig08_12.cpp(18) :
    error C2166: l-value specifies const object
```

Mensagem de erro do compilador GNU C++:

```
fig08_12.cpp: In function `void f(const int*)':
fig08_12.cpp:18: error: assignment of read-only location
```

Figura 8.12 Tentando modificar dados por meio de um ponteiro não constante para dados constantes.

(continuação)



Dica de desempenho 8.1

Se eles não precisarem ser modificados pela função chamada, passe os objetos grandes utilizando ponteiros para dados constantes ou referências para dados constantes, para obter os benefícios de desempenho da passagem por referência.



Observação de engenharia de software 8.3

Passe objetos grandes utilizando ponteiros para dados constantes, ou referências para dados constantes, para obter a segurança da passagem por valor.

Ponteiro constante para dados não constantes

Um **ponteiro constante para dados não constantes** é um ponteiro que sempre aponta para a mesma posição da memória; os dados nessa posição podem ser modificados pelo ponteiro. Esse é o padrão de um nome de array. Um nome de array é um ponteiro constante para o começo do array. Todos os dados no array podem ser acessados e alterados utilizando o nome do array e o subscrito do array. Um ponteiro constante para dados não constantes pode ser utilizado para receber um array como um argumento para uma função que acessa elementos do array utilizando a notação de subscrito do array. Os ponteiros que são declarados `const` devem ser inicializados quando são declarados. (Se o ponteiro for um parâmetro de função, ele será inicializado com um ponteiro que é passado para a função.) O programa da Figura 8.13 tenta modificar um ponteiro constante. A linha 11 declara o ponteiro `ptr` para ser do tipo `int * const`. A declaração na figura é lida da direita para a esquerda como ‘`ptr` é um ponteiro constante para um inteiro não constante’. O ponteiro é inicializado

```
1 // Figura 8.13: fig08_13.cpp
2 // Tentar modificar um ponteiro constante para dados não constantes.
3
4 int main()
5 {
6     int x, y;
7
8     // ptr é um ponteiro constante para um inteiro que pode
9     // ser modificado por ptr, mas ptr sempre aponta para a
10    // mesma posição da memória.
11    int * const ptr = &x; // ponteiro const deve ser inicializado
12
13    *ptr = 7; // permitido: *ptr não é const
14    ptr = &y; // erro: ptr é constante; não é possível atribuí-lo a um novo endereço
15    return 0; // indica terminação bem-sucedida
16 }
```

Figura 8.13 Tentando modificar um ponteiro constante para dados não constantes.

(continua)

Mensagem de erro do compilador de linha de comando Borland C++:

```
Error E2024 fig08_13.cpp 14: Cannot modify a const object in function main()
```

Mensagem de erro do compilador Microsoft Visual C++:

```
c:\cpphttp5e_examples\ch08\Fig08_13\fig08_13.cpp(14) : error C2166:
l-value specifies const object
```

Mensagem de erro do compilador GNU C++:

```
fig08_13.cpp: In function `int main()':
fig08_13.cpp:14: error: assignment of read-only variable `ptr'
```

Figura 8.13 Tentando modificar um ponteiro constante para dados não constantes.

(continuação)

com o endereço de variável de inteiro `x`. A linha 14 tenta atribuir o endereço de `y` a `ptr`, mas o compilador gera uma mensagem de erro. Observe que nenhum erro ocorre quando a linha 13 atribui o valor 7 a `*ptr` — o valor não-constante para o qual `ptr` aponta pode ser modificado utilizando o `ptr` desreferenciado, mesmo que o próprio `ptr` tenha sido declarado como `const`.



Erro comum de programação 8.6

Não inicializar um ponteiro que é declarado como `const` é um erro de compilação.

Ponteiro constante para dados constantes

A menor quantidade de privilégio de acesso é concedida por um **ponteiro constante para dados constantes**. Esse ponteiro sempre aponta para a mesma posição da memória, e os dados nessa posição da memória não podem ser modificados utilizando o ponteiro. Essa é a maneira como um array deve ser passado para uma função que somente lê o array, usando a notação de subscrito de array, e que não modifica o array. O programa da Figura 8.14 declara variável ponteiro `ptr` para ser do tipo `const int * const` (linha 14). Essa declaração é lida da direita para a esquerda como ‘`ptr` é um ponteiro constante para um inteiro constante’. A figura mostra as mensagens de erro geradas quando se tenta modificar os dados para os quais `ptr` aponta (linha 18) e quando se tenta modificar o endereço armazenado na variável ponteiro (linha 19). Observe que não ocorre nenhum erro quando o programa tenta desreferenciar `ptr` ou gerar saída do valor para o qual `ptr` aponta (linha 16), porque nem o ponteiro nem os dados para os quais ele aponta estão sendo modificados nessa instrução.

8.6 Classificação por seleção utilizando passagem por referência

Nesta seção, definimos um programa de classificação para demonstrar a passagem de arrays e elementos de array individuais por referência. Utilizamos o algoritmo de **classificação por seleção**, que é um algoritmo de classificação fácil de programar, mas, infelizmente, ineficiente. A primeira iteração do algoritmo seleciona o menor elemento no array e o troca pelo primeiro elemento. A segunda iteração seleciona o segundo menor elemento (que é o menor dos elementos restantes) e o troca pelo segundo elemento. O algoritmo continua até que a última iteração selecione o segundo maior elemento e permuta-o pelo penúltimo índice, deixando o maior elemento no último índice. Depois da i -ésima iteração, os i menores itens do array serão classificados pela ordem crescente nos primeiros i elementos do array.

Como um exemplo, considere o array

```
34 56 4 10 77 51 93 30 5 52
```

Um programa que implementa classificação por seleção primeiro determina o menor elemento (4) desse array, que está contido no elemento 2. O programa troca o 4 pelo elemento 0 (34), resultando em

```
4 56 34 10 77 51 93 30 5 52
```

[Nota: Utilizamos negrito para destacar os valores que foram trocados.] O programa então determina o menor valor dos elementos restantes (todos os elementos exceto 4), que é 5, contido no elemento 8. O programa troca o 5 pelo elemento 1 (56), resultando em

```
4 5 34 10 77 51 93 30 56 52
```

Na terceira iteração, o programa determina o próximo menor valor (10) e o troca pelo elemento 2 (34).

```
4 5 10 34 77 51 93 30 56 52
```

O processo continua até que o array seja completamente classificado.

```
4 5 10 30 34 51 52 56 77 93
```

```

1 // Figura 8.14: fig08_14.cpp
2 // Tentando modificar um ponteiro constante para dados constantes.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     int x = 5, y;
10
11     // ptr é um ponteiro constante para um inteiro constante.
12     // ptr sempre aponta para a mesma posição; o inteiro
13     // nessa posição não pode ser modificado.
14     const int *const ptr = &x;
15
16     cout << *ptr << endl;
17
18     *ptr = 7; // erro: *ptr é const; não é possível atribuir novo valor
19     ptr = &y; // error: ptr é const; não é possível atribuir endereço
20     return 0; // indica terminação bem-sucedida
21 } // fim de main

```

Mensagem de erro do compilador de linha de comando Borland C++:

```

Error E2024 fig08_14.cpp 18: Cannot modify a const object in function main()
Error E2024 fig08_14.cpp 19: Cannot modify a const object in function main()

```

Mensagem de erro do compilador Microsoft Visual C++:

```

c:\cpphttp5e_examples\ch08\Fig08_14\fig08_14.cpp(18) : error C2166:
l-value specifies const object
c:\cpphttp5e_examples\ch08\Fig08_14\fig08_14.cpp(19) : error C2166:
l-value specifies const object

```

Mensagem de erro do compilador GNU C++:

```

fig08_14.cpp: In function `int main()':
fig08_14.cpp:18: error: assignment of read-only location
fig08_14.cpp:19: error: assignment of read-only variable `ptr'

```

Figura 8.14 Tentando modificar um ponteiro constante para dados constantes.

Observe que, depois da primeira iteração, o menor elemento estará na primeira posição. Depois da segunda iteração, os dois menores elementos estarão na ordem nas duas primeiras posições. Depois da terceira iteração, os três menores elementos estarão na ordem nas três primeiras posições.

A Figura 8.15 implementa a classificação por seleção utilizando duas funções — `selectionSort` e `swap`. A função `selectionSort` (linhas 36–53) classifica o array. A linha 38 declara a variável `smallest`, que armazenará o índice do menor elemento no array restante. As linhas 41–52 iteram `size - 1` vezes. A linha 43 configura o índice do menor elemento para o índice atual. As linhas 46–49 iteram pelos elementos restantes no array. Para cada um desses elementos, a linha 48 compara seu valor com o valor do menor elemento. Se o elemento atual for menor que o menor elemento, a linha 49 atribui o índice do elemento atual a `smallest`. Quando esse loop termina, `smallest` conterá o índice do menor elemento no array restante. A linha 51 chama a função `swap` (linhas 57–62) para colocar o menor elemento restante no próximo local no array (isto é, troca os elementos do array `array[ i ]` e `array[ smallest ]`).

Façamos agora um exame mais minucioso na função `swap`. Lembre-se de que o C++ impõe o ocultamento de informações entre funções, por isso `swap` não tem acesso aos elementos de array individuais em `selectionSort`. Como `selectionSort` quer que `swap` tenha acesso aos elementos do array a serem trocados, `selectionSort` passa cada um desses elementos para `swap` por referência — o endereço de cada elemento de array é passado explicitamente. Embora os arrays inteiros sejam passados por referência, os elementos do array individuais são escalares e comumente passados por valor. Portanto, `selectionSort` utiliza o operador de endereço (&) em cada elemento

do array na chamada a swap (linha 51) para produzir a passagem por referência. A função swap (linhas 57–62) recebe &array[i] na variável ponteiro element1Ptr. O ocultamento de informações impede que swap ‘saiba’ o nome array[i], mas swap pode utilizar *element1Ptr como um sinônimo de array[i]. Portanto, quando swap referencia *element1Ptr, ele na realidade está referenciando array[i] em selectionSort. De maneira semelhante, quando swap referencia *element2Ptr, na realidade está referenciando array[smallest] em selectionSort.

Ainda que swap não tenha permissão de utilizar as instruções

```
hold = array[ i ];
array[ i ] = array[ smallest ];
array[ smallest ] = hold;
```

precisamente o mesmo efeito é alcançado por

```
int hold = *element1Ptr;
*element1Ptr = *element2Ptr;
*element2Ptr = hold;
```

na função swap da Figura 8.15.

```
1 // Figura 8.15: fig08_15.cpp
2 // Este programa coloca valores em um array, classifica os valores em
3 // ordem crescente e imprime o array resultante.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include <iomanip>
9 using std::setw;
10
11 void selectionSort( int * const, const int ); // protótipo
12 void swap( int * const, int * const ); // protótipo
13
14 int main()
15 {
16     const int arraySize = 10;
17     int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
18
19     cout << "Data items in original order\n";
20
21     for ( int i = 0; i < arraySize; i++ )
22         cout << setw( 4 ) << a[ i ];
23
24     selectionSort( a, arraySize ); // classifica o array
25
26     cout << "\nData items in ascending order\n";
27
28     for ( int j = 0; j < arraySize; j++ )
29         cout << setw( 4 ) << a[ j ];
30
31     cout << endl;
32     return 0; // indica terminação bem-sucedida
33 } // fim de main
34
35 // função para classificar um array
36 void selectionSort( int * const array, const int size )
37 {
38     int smallest; // índice do menor elemento
39
```

Figura 8.15 Classificação por seleção com passagem por referência.

(continua)

```

40 // itera sobre size - 1 elementos
41 for ( int i = 0; i < size - 1; i++ )
42 {
43     smallest = i; // primeiro índice do array remanescente
44
45     // faz um loop para localizar o índice do menor elemento
46     for ( int index = i + 1; index < size; index++ )
47
48         if ( array[ index ] < array[ smallest ] )
49             smallest = index;
50
51     swap( &array[ i ], &array[ smallest ] );
52 } // fim do for
53 } // fim da função selectionSort
54
55 // troca os valores nas posições da memória para as quais
56 // element1Ptr e element2Ptr apontem
57 void swap( int * const element1Ptr, int * const element2Ptr )
58 {
59     int hold = *element1Ptr;
60     *element1Ptr = *element2Ptr;
61     *element2Ptr = hold;
62 } // fim da função swap

```

```

Data items in original order
2  6  4  8 10 12 89 68 45 37
Data items in ascending order
2  4  6  8 10 12 37 45 68 89

```

Figura 8.15 Classificação por seleção com passagem por referência.

(continuação)

Vários recursos da função `selectionSort` devem ser observados. O cabeçalho de função (linha 36) declara `array` como `int * const array`, em vez de `int array[]`, para indicar que a função `selectionSort` recebe um array unidimensional como um argumento. Tanto o ponteiro do parâmetro `array` como o parâmetro `size` são declarados como `const` para impor o princípio do menor privilégio. Embora o parâmetro `size` receba uma cópia de um valor em `main` e modificar a cópia não possa alterar o valor em `main`, `selectionSort` não precisa alterar `size` para realizar sua tarefa — o tamanho de `array` permanece fixo durante a execução de `selectionSort`. Portanto, `size` é declarado como `const` para assegurar que ele não seja modificado. Se o tamanho do array fosse modificado durante o processo de classificação, o algoritmo de classificação não teria executado corretamente.

Observe que a função `selectionSort` recebe o tamanho do array como um parâmetro, porque a função deve ter essas informações para classificar o array. Quando um array baseado em ponteiro é passado para uma função, somente o endereço de memória do primeiro elemento do array é recebido pela função; o tamanho do array deve ser passado separadamente para a função.

Definindo a função `selectionSort` para receber o tamanho de array como um parâmetro, permitimos que a função seja utilizada por qualquer programa que classifique arrays `int` unidimensionais de tamanho arbitrário. O tamanho do array poderia ter sido programado diretamente na função, mas isso restringiria a função a processar um array de um tamanho específico e reduziria a reusabilidade da função — somente os programas que processam arrays unidimensionais `int` de um tamanho específico codificados diretamente [*‘hard coded’*] na função poderiam utilizar a função.



Observação de engenharia de software 8.4

Ao passar um array para uma função, passe também o tamanho do array (em vez de construir na função o conhecimento sobre o tamanho de array). Isso torna a função mais reutilizável.

8.7 Operadores sizeof

O C++ fornece o operador unário `sizeof` para determinar o tamanho de um array (ou de qualquer outro tipo de dados, variável ou constante) em bytes durante a compilação de programa. Quando aplicado ao nome de um array, como na Figura 8.16 (linha 14), o operador `sizeof` retorna o número total de bytes no array como um valor de tipo `size_t` (um alias para `unsigned int` na maioria dos

compiladores). Observe que esse é diferente do `size` de um `vector< int >`, por exemplo, que é o número de elementos do tipo inteiro no `vector`. O computador que utilizamos para compilar esse programa armazena variáveis do tipo `double` em 8 bytes de memória, e `array` é declarado para ter 20 elementos (linha 12), portanto `array` utiliza 160 bytes na memória. Quando aplicado a um parâmetro de ponteiro (linha 24) em uma função que recebe um array como um argumento, o operador `sizeof` retorna o tamanho do ponteiro em bytes (4), não o tamanho do array.



Erro comum de programação 8.7

Utilizar o operador `sizeof` em uma função para localizar o tamanho em bytes de um parâmetro de array resulta no tamanho em bytes de um ponteiro, não no tamanho em bytes do array.

[Nota: Quando o compilador Borland C++ é utilizado para compilar a Figura 8.16, o compilador gera a mensagem de advertência "Parameter 'ptr' is never used in function getSize(double *)". Esse aviso ocorre porque `sizeof` é na realidade um operador em tempo de compilação; portanto, a variável `ptr` não é utilizada no corpo da função em tempo de execução. Muitos compiladores emitem avisos como esse para que se saiba que uma variável não está sendo utilizada de modo que você possa removê-la de seu código ou modificar seu código para utilizar a variável adequadamente. Mensagens semelhantes ocorrem na Figura 8.17 com vários compiladores.]

O número de elementos em um array também pode ser determinado utilizando os resultados de duas operações `sizeof`. Por exemplo, considere a seguinte declaração de array:

```
double realArray[ 22 ];
```

Se variáveis do tipo de dados `double` forem armazenadas em oito bytes de memória, o array `realArray` conterá um total de 176 bytes. Para determinar o número de elementos no array, a seguinte expressão pode ser utilizada:

```
sizeof realArray / sizeof( double ) // calcula o número de elementos
```

A expressão determina o número de bytes no array `realArray` (176) e divide esse valor pelo número de bytes utilizados na memória para armazenar um valor `double` (8); o resultado é o número de elementos em `realArray` (22).

```
1 // Figura 8.16: fig08_16.cpp
2 // Operador Sizeof quando utilizado em um nome de array
3 // retorna o número de bytes no array.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 size_t getSize( double * ); // protótipo
9
10 int main()
11 {
12     double array[ 20 ]; // 20 doubles; o que ocupa 160 bytes em nosso sistema
13
14     cout << "The number of bytes in the array is " << sizeof( array );
15
16     cout << "\nThe number of bytes returned by getSize is "
17         << getSize( array ) << endl;
18     return 0; // indica terminação bem-sucedida
19 } // fim de main
20
21 // retorna o tamanho de ptr
22 size_t getSize( double *ptr )
23 {
24     return sizeof( ptr );
25 } // fim da função getSize
```

```
The number of bytes in the array is 160
The number of bytes returned by getSize is 4
```

Figura 8.16 O operador `sizeof` quando aplicado a um nome de array retorna o número de bytes no array.

Determinando os tamanhos dos tipos fundamentais, um array e um ponteiro

O programa da Figura 8.17 utiliza o operador `sizeof` para calcular o número de bytes utilizados para armazenar a maioria dos tipos de dados padrão. Note que, na saída, os tipos `double` e `long double` têm o mesmo tamanho. Os tipos podem ter tamanhos diferentes com base no sistema em que o programa é executado. Em outro sistema, por exemplo, `double` e `long double` podem ser definidos com tamanhos diferentes.

```

1 // Figura 8.17: fig08_17.cpp
2 // Demonstrando o operador sizeof.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     char c; // variável de tipo char
10    short s; // variável de tipo short
11    int i; // variável de tipo int
12    long l; // variável de tipo long
13    float f; // variável de tipo float
14    double d; // variável de tipo double
15    long double ld; // variável de tipo long double
16    int array[ 20 ]; // array de int
17    int *ptr = array; // variável de tipo int *
18
19    cout << "sizeof c = " << sizeof c
20         << "\tsizeof(char) = " << sizeof( char )
21         << "\nsizeof s = " << sizeof s
22         << "\tsizeof(short) = " << sizeof( short )
23         << "\nsizeof i = " << sizeof i
24         << "\tsizeof(int) = " << sizeof( int )
25         << "\nsizeof l = " << sizeof l
26         << "\tsizeof(long) = " << sizeof( long )
27         << "\nsizeof f = " << sizeof f
28         << "\tsizeof(float) = " << sizeof( float )
29         << "\nsizeof d = " << sizeof d
30         << "\tsizeof(double) = " << sizeof( double )
31         << "\nsizeof ld = " << sizeof ld
32         << "\tsizeof(long double) = " << sizeof( long double )
33         << "\nsizeof array = " << sizeof array
34         << "\nsizeof ptr = " << sizeof ptr << endl;
35    return 0; // indica terminação bem-sucedida
36 } // fim de main

```

```

sizeof c = 1    sizeof(char) = 1
sizeof s = 2    sizeof(short) = 2
sizeof i = 4    sizeof(int) = 4
sizeof l = 4    sizeof(long) = 4
sizeof f = 4    sizeof(float) = 4
sizeof d = 8    sizeof(double) = 8
sizeof ld = 8   sizeof(long double) = 8
sizeof array = 80
sizeof ptr = 4

```

Figura 8.17 Operador `sizeof` utilizado para determinar tamanhos de tipo de dados padrão.



Dica de portabilidade 8.3

O número de bytes utilizado para armazenar um tipo de dados particular pode variar entre sistemas. Ao escrever programas que dependem de tamanhos de tipo de dados e que executarão em vários sistemas de computador, utilize `sizeof` para determinar o número de bytes utilizados para armazenar os tipos de dados.

O operador `sizeof` pode ser aplicado a qualquer nome de variável, nome de tipo ou valor constante. Quando `sizeof` é aplicado a um nome de variável (que não seja um nome de array) ou um valor constante, o número de bytes utilizados para armazenar o tipo de variável específica ou constante é retornado. Observe que os parênteses utilizados com `sizeof` só são requeridos se o nome de um tipo (por exemplo, `int`) for fornecido como seu operando. Os parênteses utilizados com `sizeof` não são requeridos quando o operando de `sizeof` for um nome de variável ou constante. Lembre-se de que `sizeof` é um operador, não uma função e que ele tem seu efeito em tempo de compilação, não em tempo de execução.



Erro comum de programação 8.8

Omitir os parênteses em uma operação `sizeof` quando o operando é um nome de tipo é um erro de compilação.



Dica de desempenho 8.2

Como `sizeof` é um operador unário em tempo de compilação, não um operador em tempo de execução, utilizar `sizeof` não influencia negativamente o desempenho da execução.



Dica de prevenção de erro 8.3

Para evitar erros associados com a omissão dos parênteses em torno do operando do operador `sizeof`, muitos programadores colocam cada operando `sizeof` entre parênteses.

8.8 Expressões e aritmética de ponteiro

Os ponteiros são operandos válidos em expressões aritméticas, expressões de atribuição e expressões de comparação. Entretanto, nem todos os operadores normalmente utilizados nessas expressões são válidos com variáveis ponteiro. Esta seção descreve os operadores que podem ter ponteiros como operandos e como esses operadores são utilizados com ponteiros.

Várias operações aritméticas podem ser realizadas em ponteiros. Um ponteiro pode ser incrementado (`++`) ou decrementado (`--`), um inteiro pode ser adicionado a um ponteiro (`+` ou `+=`), um inteiro pode ser subtraído de um ponteiro (`-` ou `-=`) ou um ponteiro pode ser subtraído de outro.

Suponha que o array `int v[ 5 ]` tenha sido declarado e que seu primeiro elemento esteja na posição da memória 3000. Suponha que o ponteiro `vPtr` tenha sido inicializado para apontar para `v[ 0 ]` (isto é, o valor de `vPtr` é 3000). A Figura 8.18 diagrama essa situação para uma máquina com inteiros de quatro bytes. Observe que `vPtr` pode ser inicializado para apontar para o array `v` com qualquer uma das seguintes instruções (porque o nome de um array é equivalente ao endereço de seu primeiro elemento):

```
int *vPtr = v;
int *vPtr = &v[ 0 ];
```



Dica de portabilidade 8.4

A maioria dos computadores hoje tem inteiros de dois bytes ou quatro bytes. Algumas máquinas mais novas utilizam inteiros de oito bytes. Como os resultados da aritmética de ponteiros dependem do tamanho dos objetos para os quais um ponteiro aponta, a aritmética de ponteiros é dependente da máquina.

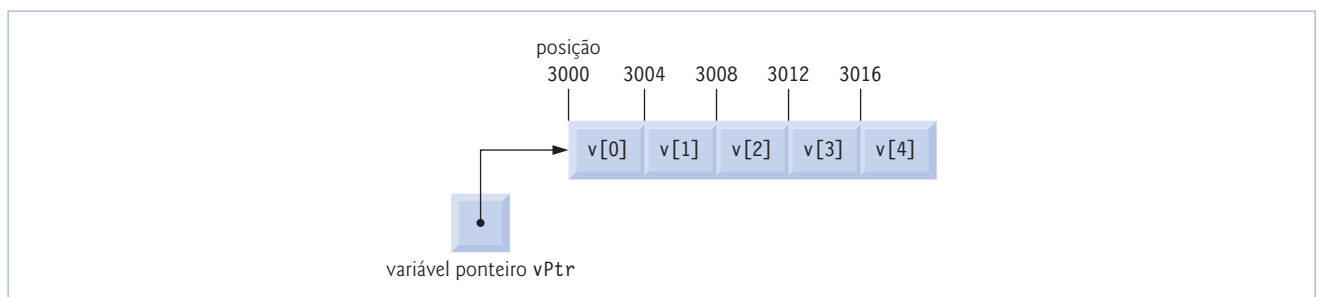


Figura 8.18 O array `v` e uma variável ponteiro `vPtr` que aponta para `v`.

Na aritmética convencional, a adição $3000 + 2$ resulta no valor 3002. Normalmente não é esse o caso com a aritmética de ponteiros. Quando um inteiro é adicionado a, ou subtraído de, um ponteiro, o ponteiro simplesmente não é incrementado ou decrementado por esse inteiro, mas por esse inteiro vezes o tamanho do objeto que o ponteiro referencia. O número de bytes depende do tipo de dados do objeto. Por exemplo, a instrução

```
vPtr += 2;
```

produziria 3008 ($3000 + 2 * 4$), supondo que um `int` é armazenado em quatro bytes de memória. No array `v`, `vPtr` agora apontaria para `v[ 2 ]` (Figura 8.19). Se um inteiro fosse armazenado em dois bytes de memória, então o cálculo precedente resultaria na posição da memória 3004 ($3000 + 2 * 2$). Se o array fosse de um tipo de dados diferente, a instrução precedente incrementaria o ponteiro por duas vezes o número de bytes que ele aceita para armazenar um objeto desse tipo de dados. Ao realizar a aritmética de ponteiros em um array de caracteres, os resultados serão consistentes com a aritmética regular, porque cada caractere tem um byte de comprimento.

Se `vPtr` tiver sido incrementado por 3016, que aponta para `v[4]`, a instrução

```
vPtr -= 4;
```

configuraria `vPtr` de volta para 3000 — o começo do array. Se um ponteiro estiver sendo incrementado ou decrementado por um, os operadores de incremento (`++`) e decremento (`--`) podem ser utilizados. Cada uma das instruções

```
++vPtr;
```

```
vPtr++;
```

incrementa o ponteiro para apontar para o próximo elemento do array. Cada uma das instruções

```
--vPtr;
```

```
vPtr--;
```

decrementa o ponteiro para apontar para o elemento anterior do array.

As variáveis ponteiro que apontam para o mesmo array podem ser subtraídas uma da outra. Por exemplo, se `vPtr` contiver a posição 3000 e `v2Ptr` contiver o endereço 3008, a instrução

```
x = v2Ptr - vPtr;
```

atribuiria a `x` o número de elementos do array de `vPtr` a `v2Ptr` — nesse caso, 2. A aritmética de ponteiros não faz sentido a menos que realizada em um ponteiro que aponta para um array. Não podemos pressupor que duas variáveis do mesmo tipo estejam armazenadas contigualmente na memória a menos que elas sejam elementos adjacentes de um array.



Erro comum de programação 8.9

Utilizar aritmética de ponteiros em um ponteiro que não referencia um array de valores é um erro de lógica.



Erro comum de programação 8.10

Subtrair ou comparar dois ponteiros que não referenciam elementos do mesmo array é um erro de lógica.



Erro comum de programação 8.11

Utilizar aritmética de ponteiros para incrementar ou decrementar um ponteiro de modo que esse ponteiro reference um elemento depois do fim, ou antes do começo do array, é normalmente um erro de lógica.

Um ponteiro pode ser atribuído a outro ponteiro se ambos forem do mesmo tipo. Caso contrário, um operador de coerção deve ser utilizado para converter o valor do ponteiro à direita da atribuição no tipo de ponteiro à esquerda da atribuição. A exceção a essa regra é o ponteiro para `void` (isto é, `void *`), que é um ponteiro genérico capaz de representar qualquer tipo de ponteiro. Um ponteiro de

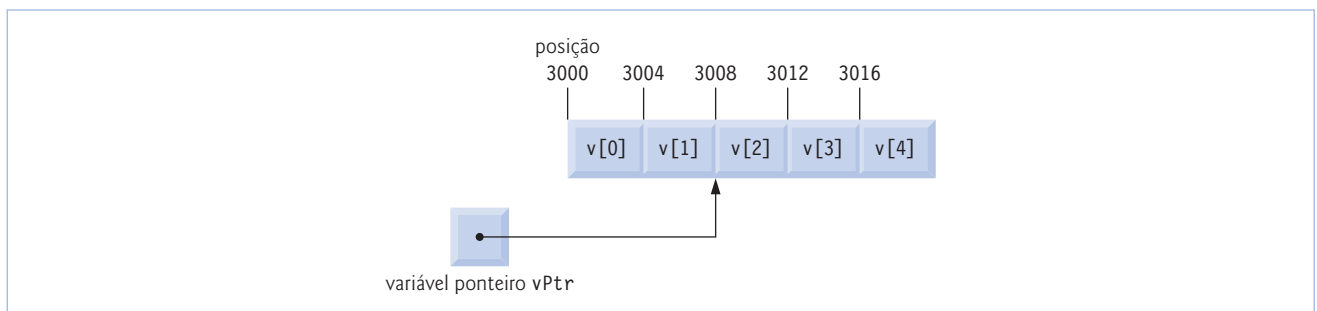


Figura 8.19 Ponteiro `vPtr` depois da aritmética de ponteiros.

tipo `void *` sem coerção pode ser atribuído a todos os tipos de ponteiro. Entretanto, um ponteiro do tipo `void *` não pode ser atribuído diretamente a um ponteiro de outro tipo — o ponteiro do tipo `void *` deve primeiro sofrer coerção para o tipo de ponteiro adequado.



Observação de engenharia de software 8.5

Os argumentos de ponteiro não-constantes podem ser passados para parâmetros de ponteiro constantes. Isso é útil quando o corpo de um programa utiliza um ponteiro não-constante para acessar dados, mas não quer que os dados sejam modificados por uma chamada de função no corpo do programa.

Um ponteiro `void *` não pode ser desreferenciado. Por exemplo, o compilador ‘sabe’ que um ponteiro para `int` referencia quatro bytes de memória em uma máquina com inteiros de quatro bytes, mas um ponteiro para `void` contém simplesmente um endereço de memória para um tipo de dados desconhecido — o número preciso de bytes que o ponteiro referencia e o tipo dos dados não são conhecidos pelo compilador. O compilador deve conhecer o tipo de dados para determinar o número de bytes a ser desreferenciado por um ponteiro particular — para um ponteiro para `void`, esse número de bytes não pode ser determinado a partir do tipo.



Erro comum de programação 8.12

Atribuir um ponteiro de um tipo a um ponteiro de outro (diferente de `void *`) sem fazer coerção do primeiro ponteiro para o tipo do segundo ponteiro é um erro de compilação.



Erro comum de programação 8.13

Todas as operações em um ponteiro `void *` são erros de compilação, exceto comparar ponteiros `void *` com outros ponteiros, fazendo coerção dos ponteiros `void *` para tipos de ponteiros válidos e atribuindo endereços a ponteiros `void *`.

Os ponteiros podem ser comparados utilizando operadores de igualdade e operadores relacionais. As comparações que utilizam operadores relacionais não têm sentido a menos que os ponteiros apontem para membros do mesmo array. As comparações de ponteiro comparam os endereços armazenados nos ponteiros. Uma comparação de dois ponteiros que apontam para o mesmo array poderia mostrar, por exemplo, que um ponteiro aponta para um elemento de número mais alto no array do que o outro ponteiro. Uma utilização comum da comparação de ponteiro é determinar se um ponteiro é 0 (isto é, o ponteiro é um ponteiro nulo — ele não aponta para nada).

8.9 Relacionamento entre ponteiros e arrays

Os arrays e os ponteiros estão intimamente relacionados em C++ e podem ser utilizados quase intercambiavelmente. É possível pensar em um nome de array como um ponteiro constante. Os ponteiros podem ser utilizados para fazer qualquer operação que envolva subscrito de array.

Suponha as seguintes declarações:

```
int b[ 5 ]; // cria array int b de 5 elementos
int *bPtr; // cria ponteiro int bPtr
```

Como o nome de array (sem subscrito) é um ponteiro (constante) para o primeiro elemento do array, podemos configurar `bPtr` como o endereço do primeiro elemento no array `b` com a instrução

```
bPtr = b; // atribui o endereço de array b ao bPtr
```

Isso é equivalente a aceitar o endereço do primeiro elemento do array como mostrado a seguir:

```
bPtr = &b[ 0 ]; // também atribui o endereço do array b ao bPtr
```

O elemento do array `b[ 3 ]` pode ser alternativamente referenciado com a expressão de ponteiro

```
*( bPtr + 3 )
```

O 3 na expressão precedente é o **deslocamento** para o ponteiro. Quando o ponteiro aponta para o começo de um array, o deslocamento indica que elemento do array deve ser referenciado e o valor de deslocamento é idêntico ao subscrito de array. A notação anterior é referida como **notação de ponteiro/deslocamento**. Os parênteses são necessários, porque a precedência de `*` é mais alta que a precedência de `+`. Sem os parênteses, a expressão acima adicionaria 3 ao valor de `*bPtr` (isto é, 3 seriam adicionados a `b[ 0 ]`, supondo que `bPtr` aponta para o início do array). Assim como o elemento do array pode ser referenciado com uma expressão de ponteiro, o endereço

```
&b[ 3 ]
```

pode ser escrito com a expressão de ponteiro

```
bPtr + 3
```

O nome do array pode ser tratado como um ponteiro e utilizado na aritmética de ponteiros. Por exemplo, a expressão

```
*( b + 3 )
```

também referencia o elemento `b[ 3 ]` do array. Em geral, todas as expressões de array subscriptadas podem ser escritas com um ponteiro e um deslocamento. Nesse caso, a notação de ponteiro/deslocamento foi utilizada com o nome do array como um ponteiro. Observe que a expressão precedente não modifica o nome do array de nenhuma maneira; `b` ainda aponta para o primeiro elemento no array.

Os ponteiros podem ser indexados com subscritos exatamente como arrays. Por exemplo, a expressão

```
bPtr[ 1 ]
```

referencia o elemento do array `b[ 1 ]`; essa expressão utiliza **notação de ponteiro/subscrito**.

Lembre-se de que um nome de array é um ponteiro constante; ele sempre aponta para o começo do array. Portanto, a expressão

```
b += 3
```

causa um erro de compilação, porque tenta modificar o valor do nome do array (uma constante) com a aritmética de ponteiros.



Erro comum de programação 8.14

Embora os nomes de array sejam ponteiros para o começo do array e os ponteiros possam ser modificados em expressões aritméticas, os nomes de array não podem ser modificados em expressões aritméticas, porque são ponteiros constantes.



Boa prática de programação 8.2

Por questão de clareza, use notação de array em vez de notação de ponteiro ao manipular arrays.

A Figura 8.20 utiliza as quatro notações discutidas nesta seção para referenciar elementos de um array — a notação de subscrito de array, a notação de ponteiro/deslocamento com o nome de array como um ponteiro, a notação de subscrito de ponteiro e a notação de ponteiro/deslocamento com um ponteiro — para realizar a mesma tarefa, isto é, imprimir os quatro elementos do array de inteiro `b`.

Para ilustrar ainda mais a intercambialidade de arrays e ponteiros, vejamos as duas funções de copiar strings — `copy1` e `copy2` — no programa da Figura 8.21. Ambas as funções copiam uma string em um array de caracteres. Depois de uma comparação dos protótipos de função para `copy1` e `copy2`, as funções parecem idênticas (por causa da intercambialidade de arrays e ponteiros). Essas funções realizam a mesma tarefa, mas são implementadas de modo diferente.

```

1 // Figura 8.20: fig08_20.cpp
2 // Utilizando notações de subscrito e de ponteiro com arrays.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9     int b[] = { 10, 20, 30, 40 }; // cria o array b de 4 elementos
10    int *bPtr = b; // configura bPtr para apontar para o array b
11
12    // gera saída do array b utilizando notação de subscrito de array
13    cout << "Array b printed with:\n\nArray subscript notation\n";
14
15    for ( int i = 0; i < 4; i++ )
16        cout << "b[" << i << "] = " << b[ i ] << '\n';
17
18    // gera saída do array b utilizando a notação de nome de array e a de ponteiro/deslocamento
19    cout << "\nPointer/offset notation where "
20        << "the pointer is the array name\n";
21
22    for ( int offset1 = 0; offset1 < 4; offset1++ )
23        cout << "*(b + " << offset1 << ") = " << *( b + offset1 ) << '\n';
24
25    // gera saída do array b utilizando bPtr e notação de subscrito de array
26    cout << "\nPointer subscript notation\n";
27

```

Figura 8.20 Referenciando elementos do array com o nome do array e com ponteiros.

(continua)

```

28     for ( int j = 0; j < 4; j++ )
29         cout << "bPtr[" << j << "] = " << bPtr[ j ] << '\n';
30
31     cout << "\nPointer/offset notation\n";
32
33     // gera saída do array b utilizando bPtr e notação de ponteiro/deslocamento
34     for ( int offset2 = 0; offset2 < 4; offset2++ )
35         cout << "*(bPtr + " << offset2 << ") = "
36             << *( bPtr + offset2 ) << '\n';
37
38     return 0; // indica terminação bem-sucedida
39 } // fim de main

```

Array b printed with:

Array subscript notation

```

b[0] = 10
b[1] = 20
b[2] = 30
b[3] = 40

```

Pointer/offset notation where the pointer is the array name

```

*(b + 0) = 10
*(b + 1) = 20
*(b + 2) = 30
*(b + 3) = 40

```

Pointer subscript notation

```

bPtr[0] = 10
bPtr[1] = 20
bPtr[2] = 30
bPtr[3] = 40

```

Pointer/offset notation

```

*(bPtr + 0) = 10
*(bPtr + 1) = 20
*(bPtr + 2) = 30
*(bPtr + 3) = 40

```

Figura 8.20 Referenciando elementos do array com o nome do array e com ponteiros.

(continuação)

A função `copy1` (linhas 26–31) utiliza a notação de subscrito de array para copiar a string em `s2` para o array de caracteres `s1`. A função declara uma variável contadora de inteiros `i` para utilizar como o subscrito de array. O cabeçalho da instrução `for` (linha 29) realiza toda a operação de cópia — seu corpo é a instrução vazia. O cabeçalho especifica que `i` é inicializado como zero e incrementado por um a cada iteração do loop. A condição no `for`, `( s1[ i ] = s2[ i ] ) != '\0'`, realiza a operação de cópia de caractere por caractere de `s2` para `s1`. Quando o caractere nulo é encontrado em `s2`, ele é atribuído a `s1` e o loop termina, porque o caractere nulo é igual a `'\0'`. Lembre-se de que o valor de uma instrução de atribuição é o valor atribuído a seu operando esquerdo.

A função `copy2` (linhas 34–39) utiliza ponteiros e aritmética de ponteiros para copiar a string em `s2` para o array de caracteres `s1`. Novamente, o cabeçalho da instrução `for` (linha 37) realiza a operação inteira de cópia. O cabeçalho não inclui nenhuma variável de inicialização. Como na função `copy1`, a condição `( *s1 = *s2 ) != '\0'` realiza a operação de cópia. O ponteiro `s2` é desreferenciado e o caractere resultante é atribuído ao ponteiro desreferenciado `s1`. Depois da atribuição na condição, o loop incrementa ambos os ponteiros, então eles apontam para o próximo elemento de array `s1` e o próximo caractere de string `s2`, respectivamente. Quando o loop encontra o caractere nulo em `s2`, o caractere nulo é atribuído ao ponteiro desreferenciado `s1` e o loop termina. Observe que a ‘parte de incremento’ dessa instrução `for` tem duas expressões de incremento separadas por um operador vírgula.

O primeiro argumento para `copy1` e `copy2` deve ser um array grande o bastante para armazenar a string no segundo argumento. Caso contrário, pode ocorrer um erro quando `for` feita uma tentativa de escrever em uma posição da memória além dos limites do array (lembre-se de que ao utilizar arrays baseados em ponteiro, não há nenhuma verificação de limite ‘integrada’). Além disso, observe que

```

1 // Figura 8.21: fig08_21.cpp
2 // Copiando uma string utilizando a notação de array e a notação de ponteiro.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 void copy1( char *, const char * ); // protótipo
8 void copy2( char *, const char * ); // protótipo
9
10 int main()
11 {
12     char string1[ 10 ];
13     char *string2 = "Hello";
14     char string3[ 10 ];
15     char string4[] = "Good Bye";
16
17     copy1( string1, string2 ); // copia string2 para string1
18     cout << "string1 = " << string1 << endl;
19
20     copy2( string3, string4 ); // copia string4 para string3
21     cout << "string3 = " << string3 << endl;
22     return 0; // indica terminação bem-sucedida
23 } // fim de main
24
25 // copia s2 para s1 utilizando notação de array
26 void copy1( char * s1, const char * s2 )
27 {
28     // a cópia ocorre no cabeçalho do for
29     for ( int i = 0; ( s1[ i ] = s2[ i ] ) != '\0'; i++ )
30         ; // não faz nada no corpo
31 } // fim da função copy1
32
33 // copia s2 para s1 utilizando notação de ponteiro
34 void copy2( char *s1, const char *s2 )
35 {
36     // a cópia ocorre no cabeçalho do for
37     for ( ; ( *s1 = *s2 ) != '\0'; s1++, s2++ )
38         ; // não faz nada no corpo
39 } // fim da função copy2

```

```

string1 = Hello
string3 = Good Bye

```

Figura 8.21 Cópia de strings utilizando a notação de array e a notação de ponteiro.

o segundo parâmetro de cada função é declarado como `const char *` (um ponteiro para um caractere constante — isto é, uma string constante). Em ambas as funções, o segundo argumento é copiado para o primeiro argumento — os caracteres são copiados um a um a partir do segundo argumento, mas os caracteres nunca são modificados. Portanto, o segundo parâmetro é declarado para apontar para um valor constante para impor o princípio do menor privilégio —nenhuma função precisa modificar o segundo argumento, então nenhuma função recebe a permissão de modificar o segundo argumento.

8.10 Arrays de ponteiros

Os arrays podem conter ponteiros. Uma utilização comum dessa estrutura de dados é formar um array de strings baseadas em ponteiro, referido simplesmente como um **array de string**. Toda entrada no array é uma string, mas em C++ uma string é essencialmente um ponteiro para seu primeiro caractere, então cada entrada em um array de strings é simplesmente um ponteiro para o primeiro caractere de uma string. Considere a declaração de array de string `string s[10]` que poderia ser útil na representação de um baralho:

```
const char *suit[ 4 ] =
    { "Hearts", "Diamonds", "Clubs", "Spades" };
```

A parte `suit[4]` da declaração indica um array de quatro elementos. A parte `const char *` da declaração indica que cada elemento de array `suit` é do tipo ‘ponteiro para dados `char` constantes’. Os quatro valores a ser colocados no array são "Hearts", "Diamonds", "Clubs" e "Spades". Cada um é armazenado na memória como uma string de caracteres terminada por caractere nulo, que é um caractere mais longo que o número de caracteres entre aspas. As quatro strings são os caracteres de tamanho sete, nove, seis e sete (incluindo seus caracteres nulos de terminação), respectivamente. Embora pareça que essas strings estão sendo colocadas no array `suit`, somente os ponteiros são realmente armazenados no array, como mostrado na Figura 8.22. Cada ponteiro aponta para o primeiro caractere de sua string correspondente. Portanto, mesmo que o array `suit` tenha tamanho fixo, ele fornece acesso a strings de caractere de qualquer comprimento. Essa flexibilidade é um exemplo das poderosas capacidades da estrutura de dados do C++.

As strings de naipe poderiam ser colocadas em um array bidimensional, em que cada linha representa um naipe e cada coluna representa uma das letras de um nome de naipe. Essa estrutura de dados deve ter um número fixo de colunas por linha, e esse número deve ser tão grande quanto a maior string. Portanto, uma quantidade considerável de memória é desperdiçada quando armazenamos um grande número de strings, das quais a maioria é mais curta que a string mais longa. Utilizamos arrays de strings para ajudar a representar um baralho na próxima seção.

Os arrays de string são comumente utilizados com **argumentos de linha de comando** que são passados para a função `main` quando um programa inicia a execução. Esses argumentos seguem o nome de programa quando um programa é executado da linha de comando. Uma utilização típica de argumentos de linha de comando é passar opções para um programa. Por exemplo, a partir da linha de comando em um computador Windows, o usuário pode digitar

```
dir /P
```

para listar o conteúdo do diretório atual e pausar depois de cada tela das informações. Quando o comando `dir` executa, a opção `/P` é passada para `dir` como um argumento de linha de comando. Esses argumentos são colocados em um array de string que `main` recebe como um argumento. Discutimos os argumentos de linha de comando no Apêndice E, “Tópicos sobre o código C legado”.

8.1.1 Estudo de caso: simulação de embaralhamento e distribuição de cartas

Esta seção utiliza a geração de números aleatórios para desenvolver um programa de simulação de embaralhamento e distribuição de cartas. Esse programa pode então ser utilizado como uma base para implementar programas que reproduzem jogos de cartas específicos. Para revelar alguns problemas de desempenho sutis, utilizamos intencionalmente algoritmos de embaralhamento e distribuição subótimos. Nos exercícios, desenvolvemos algoritmos mais eficientes.

Utilizando a abordagem de refinamento passo a passo de cima para baixo, desenvolvemos um programa que irá embaralhar um baralho de 52 cartas e, em seguida, negociar cada uma delas. A abordagem de cima para baixo é particularmente útil para atacar problemas maiores e mais complexos que vimos nos capítulos anteriores.

Utilizamos um array bidimensional 4 por 13 `deck` para representar o baralho (Figura 8.23). As linhas correspondem aos naipes — a linha 0 corresponde ao naipe de copas, a linha 1 ao de ouros, a linha 2 ao de paus e a linha 3 ao de espadas. As colunas correspondem aos valores de face das cartas — as colunas de 0 a 9 correspondem às faces de ás a 10, respectivamente, e as colunas de 10 a 12 correspondem ao valete, dama e rei, respectivamente. Carregaremos o array de strings `suit` com strings de caracteres que representam os quatro naipes (como na Figura 8.22) e o array de strings `face` com strings de caracteres que representam os 13 valores de face.

Esse baralho simulado pode ser embaralhado como mostrado a seguir. Primeiro o array `deck` é inicializado como zeros. Em seguida, uma `row` (0–3) e uma `column` (0–12) são escolhidas aleatoriamente. O número 1 é inserido no elemento do array `deck[ row ][ column ]` para indicar que essa será a primeira carta distribuída do conjunto de cartas embaralhadas. Esse processo continua com os números 2, 3, ..., 52 sendo inseridos aleatoriamente no array `deck` para indicar as cartas que devem ser colocadas em segundo lugar, terceiro lugar, ... e 52º lugar no baralho embaralhado. À medida que o array `deck` começa a ser preenchido com os números de carta, é possível que uma carta seja selecionada duas vezes (isto é, `deck[ row ][ column ]` será não-zero quando for selecionada). Essa seleção simplesmente é ignorada, e outras `rows` e `columns` são repetidamente escolhidas de forma aleatória até que uma carta não selecionada seja localizada. Por fim, os números 1 a 52 ocuparão os 52 slots do array `deck`. Nesse ponto, o baralho está completamente embaralhado.

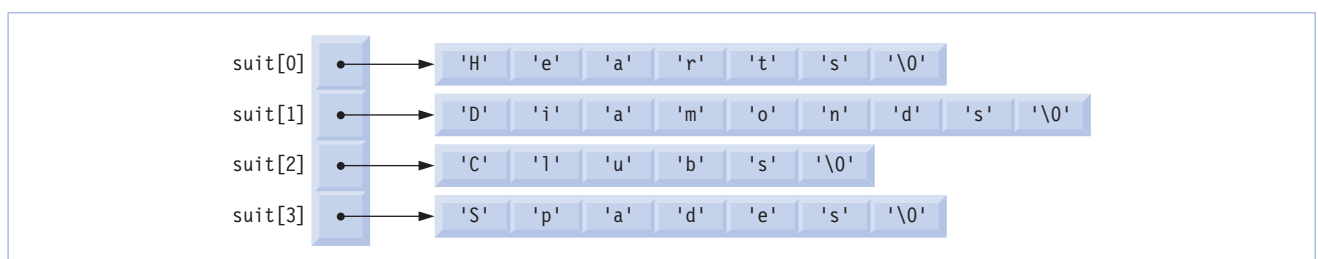


Figura 8.22 Representação gráfica do array `suit`.

		Ace	Two	Three	Four	Five	Six	Seven	Eight	Nine	Ten	Jack	Queen	King
		0	1	2	3	4	5	6	7	8	9	10	11	12
Hearts	0													
Diamonds	1													
Clubs	2													
Spades	3													

deck[2][12] representa o rei de paus (King of Clubs)

Clubs King

Figura 8.23 A representação de array bidimensional de um baralho.

Esse algoritmo de embaralhamento poderia executar por um período indefinidamente longo se as cartas que já foram embaralhadas fossem repetidamente selecionadas a esmo. Esse fenômeno é conhecido como **adiamento indefinido** (também chamado **inanição**). Nos exercícios, discutimos um algoritmo de embaralhamento melhor que elimina a possibilidade de adiamiento indefinido.



Dica de desempenho 8.3

Às vezes os algoritmos que emergem de modo ‘natural’ podem conter problemas de desempenho sutis como o adiamiento indefinido. Busque algoritmos que evitem adiamiento indefinido.

Para distribuir a primeira carta, procuramos no array o elemento `deck[ row ][ column ]` que corresponda a 1. Isso é realizado com uma instrução aninhada `for` que varia `row` de 0 a 3 e `column` de 0 a 12. A que carta esse slot de array corresponde? O array `suit` foi pré-carregado com os quatro naipes, portanto, para obter o naipe, imprimimos a string de caracteres `suit[ row ]`. De modo semelhante, para obter o valor de face da carta, imprimimos a string de caracteres `face[ column ]`. Imprimimos também a string de caracteres " of ". Imprimir essas informações na ordem adequada permite imprimir cada carta na forma "King of Clubs" [Rei de paus], "Ace of Diamonds" [Ás de ouros] e assim por diante.

Vamos prosseguir com o processo de refinamento passo a passo de cima para baixo. A parte superior é simplesmente

Embaralhe e distribua as 52 cartas

Nosso primeiro refinamento resulta em

Inicialize o array `suit`

Inicialize o array `face`

Inicialize o array `deck`

Embaralhe as cartas

Distribua as 52 cartas

‘Embaralhe as cartas’ pode ser expandido como mostrado a seguir:

Para cada uma das 52 cartas

Coloque o número da carta no slot de baralho vazio selecionado aleatoriamente

‘Distribua as 52 cartas’ pode ser expandido como mostrado a seguir:

Para cada uma das 52 cartas

Localize o número no array `deck` e imprima a face e o naipe da carta

Incorporar essas expansões produz nosso segundo refinamento completo:

Inicialize o array `suit`

Inicialize o array `face`

Inicialize o array `deck`

Para cada uma das 52 cartas

Coloque o número da carta no slot vazio selecionado aleatoriamente

Para cada uma das 52 cartas

Localize o número no array `deck` e imprima a face e o naipe da carta

‘Coloque o número de carta no slot vazio selecionado aleatoriamente’ pode ser expandido como mostrado a seguir:

Escolha o slot aleatoriamente

Enquanto o slot escolhido tiver sido previamente escolhido

Escolha o slot aleatoriamente

Coloque o número de carta no slot escolhido do baralho

‘Localize o número de carta no array deck e imprima a face e o naipe da carta’ pode ser expandido como mostrado a seguir:

Para cada slot do array deck
Se o slot contiver o número da carta
Imprima a face e o naipe da carta

Incorporar essas expansões produz nosso terceiro refinamento (Figura 8.24).

Isso completa o processo de refinamento. As figuras 8.25–8.27 contêm o programa de embaralhamento e distribuição de cartas e uma execução de exemplo. As linhas 61–67 da função `deal` (Figura 8.26) implementam as linhas 1–2 da Figura 8.24. O construtor (linhas 22–35 da Figura 8.26) implementa as linhas 1–3 da Figura 8.24. A função `shuffle` (linhas 38–55 da Figura 8.26) implementa as linhas 5–11 da Figura 8.24. A função `deal` (linhas 58–88 da Figura 8.26) implementa as linhas 13–16 da Figura 8.24. Observe a formatação de saída utilizada na função `deal` (linhas 81–83 da Figura 8.26). A instrução de saída gera saída da face alinhada à direita em um campo de cinco caracteres e gera saída do naipe alinhado à esquerda em um campo de oito caracteres (Figura 8.27). A saída é impressa em formato de duas colunas — se a carta cuja saída está sendo gerada estiver na primeira coluna, uma tabulação é enviada para a saída depois da carta a ser movida para a segunda coluna (linha 83); caso contrário, uma nova linha é enviada para a saída.

Há também uma fraqueza no algoritmo de distribuição de cartas. Uma vez que uma correspondência é encontrada, mesmo que seja encontrada na primeira tentativa, as duas instruções `for` internas continuam procurando uma correspondência nos elementos restantes de deck. Nos exercícios, corrigimos essa deficiência.

```

1  Inicialize o array suit
2  Inicialize o array face
3  Inicialize o array deck
4
5  Para cada uma das 52 cartas
6      Escolha o slot aleatoriamente
7
8      Enquanto o slot tiver sido previamente escolhido
9          Escolha o slot aleatoriamente
10
11     Coloque o número de carta no slot escolhido do baralho
12
13  Para cada uma das 52 cartas
14      Para cada slot do array deck
15          Se o slot contiver o número de carta desejado
16              Imprima a face e o naipe da carta

```

Figura 8.24 O algoritmo em pseudocódigo para o programa de embaralhamento e distribuição de cartas.

```

1  // Figura 8.25: DeckOfCards.h
2  // Definição da classe DeckOfCards que
3  // representa um baralho.
4
5  // Definição da classe DeckOfCards
6  class DeckOfCards
7  {
8  public:
9      DeckOfCards(); // construtor inicializa deck
10     void shuffle(); // embaralha as cartas do baralho
11     void deal(); // distribui as cartas do baralho
12 private:
13     int deck[ 4 ][ 13 ]; // representa o baralho de cartas
14 }; // fim da classe DeckOfCards

```

Figura 8.25 Arquivo de cabeçalho `DeckOfCards`.

```

1 // Figura 8.26: DeckOfCards.cpp
2 // Definições de função-membro para a classe DeckOfCards que simula
3 // o embaralhamento e distribuição de um baralho.
4 #include <iostream>
5 using std::cout;
6 using std::left;
7 using std::right;
8
9 #include <iomanip>
10 using std::setw;
11
12 #include <cstdlib> // protótipos para rand e srand
13 using std::rand;
14 using std::srand;
15
16 #include <ctime> // protótipo para time
17 using std::time;
18
19 #include "DeckOfCards.h" // definição da classe DeckOfCards
20
21 // construtor-padrão DeckOfCards inicializa deck
22 DeckOfCards::DeckOfCards()
23 {
24     // itera pelas linhas do baralho
25     for ( int row = 0; row <= 3; row++ )
26     {
27         // itera pelas colunas do baralho para linha atual
28         for ( int column = 0; column <= 12; column++ )
29         {
30             deck[ row ][ column ] = 0; // inicializa slot de deck como 0
31         } // fim do for interno
32     } // fim do for externo
33
34     srand( time( 0 ) ); // semeia o gerador de número aleatório
35 } // fim do construtor-padrão DeckOfCards
36
37 // embaralha as cartas do baralho
38 void DeckOfCards::shuffle()
39 {
40     int row; // representa o valor do naipe da carta
41     int column; // representa o valor da face da carta
42
43     // para cada uma das 52 cartas, escolhe um slot aleatoriamente
44     for ( int card = 1; card <= 52; card++ )
45     {
46         do // escolhe uma nova localização aleatória até um slot vazio ser encontrado
47         {
48             row = rand() % 4; // seleciona a linha aleatoriamente
49             column = rand() % 13; // seleciona a coluna aleatoriamente
50         } while( deck[ row ][ column ] != 0 ); // fim da instrução do...while
51
52         // coloca o número de carta no slot escolhido
53         deck[ row ][ column ] = card;
54     } // fim do for
55 } // fim da função shuffle
56
57 // distribui as cartas do baralho

```

Figura 8.26 Definições de funções-membro para embaralhamento e distribuição.

(continua)

```

58 void DeckOfCards::deal()
59 {
60     // inicializa o array suit
61     static const char *suit[ 4 ] =
62     { "Hearts", "Diamonds", "Clubs", "Spades" };
63
64     // inicializa o array face
65     static const char *face[ 13 ] =
66     { "Ace", "Deuce", "Three", "Four", "Five", "Six", "Seven",
67       "Eight", "Nine", "Ten", "Jack", "Queen", "King" };
68
69     // para cada uma das 52 cartas
70     for ( int card = 1; card <= 52; card++ )
71     {
72         // itera pelas linhas do baralho
73         for ( int row = 0; row <= 3; row++ )
74         {
75             // itera pelas colunas de baralho para linha atual
76             for ( int column = 0; column <= 12; column++ )
77             {
78                 // se o slot contiver a carta atual, exibe a carta
79                 if ( deck[ row ][ column ] == card )
80                 {
81                     cout << setw( 5 ) << right << face[ column ]
82                        << " of " << setw( 8 ) << left << suit[ row ]
83                        << ( card % 2 == 0 ? '\n' : '\t' );
84                 } // fim do if
85             } // fim do for mais interno
86         } // fim do for interno
87     } // fim do for externo
88 } // fim da função deal

```

Figura 8.26 Definições de funções-membro para embaralhamento e distribuição.

(continuação)

```

1 // Figura 8.27: fig08_27.cpp
2 // Programa de embaralhamento e distribuição de cartas.
3 #include "DeckOfCards.h" // Definição da classe DeckOfCards
4
5 int main()
6 {
7     DeckOfCards deckOfCards; // cria objeto DeckOfCards
8
9     deckOfCards.shuffle(); // embaralha as cartas
10    deckOfCards.deal(); // distribui as cartas
11    return 0; // indica terminação bem-sucedida
12 } // fim de main

```

Nine of Spades	Seven of Clubs
Five of Spades	Eight of Clubs
Queen of Diamonds	Three of Hearts
Jack of Spades	Five of Diamonds
Jack of Diamonds	Three of Diamonds
Three of Clubs	Six of Clubs
Ten of Clubs	Nine of Diamonds

Figura 8.27 Programa de embaralhamento e distribuição de cartas.

(continua)

Ace of Hearts	Queen of Hearts
Seven of Spades	Deuce of Spades
Six of Hearts	Deuce of Clubs
Ace of Clubs	Deuce of Diamonds
Nine of Hearts	Seven of Diamonds
Six of Spades	Eight of Diamonds
Ten of Spades	King of Hearts
Four of Clubs	Ace of Spades
Ten of Hearts	Four of Spades
Eight of Hearts	Eight of Spades
Jack of Hearts	Ten of Diamonds
Four of Diamonds	King of Diamonds
Seven of Hearts	King of Spades
Queen of Spades	Four of Hearts
Nine of Clubs	Six of Diamonds
Deuce of Hearts	Jack of Clubs
King of Clubs	Three of Spades
Queen of Clubs	Five of Clubs
Five of Hearts	Ace of Diamonds

Figura 8.27 Programa de embaralhamento e distribuição de cartas.

(continuação)

8.12 Ponteiros de função

Um ponteiro para uma função contém o endereço da função na memória. No Capítulo 7, vimos que o nome de um array é, na realidade, o endereço na memória do primeiro elemento do array. De maneira semelhante, o nome de uma função é o endereço inicial na memória do código que realiza a tarefa da função. Os ponteiros para funções podem ser passados para funções, retornados de funções, armazenados em arrays e atribuídos a outros ponteiros de função.

Classificação por seleção para múltiplos propósitos utilizando ponteiros de função

Para ilustrar o uso de ponteiros para funções, a Figura 8.28 modifica o programa de classificação por seleção da Figura 8.15. A Figura 8.28 consiste em `main` (linhas 17–55) e nas funções `selectionSort` (linhas 59–76), `swap` (linhas 80–85), `ascending` (linhas 89–92) e `descending` (linhas 96–99). A função `selectionSort` recebe um ponteiro para uma função — uma função `ascending` ou `descending` — como um argumento além do array de inteiros a classificar e o tamanho do array. As funções `ascending` e `descending` determinam a ordem da classificação. O programa pede para o usuário escolher se o array deve ser classificado em ordem crescente ou decrescente (linhas 24–26). Se a entrada do usuário for 1, um ponteiro para a função `ascending` é passado para a função `selectionSort` (linha 37), fazendo com que o array seja classificado na ordem crescente. Se a entrada do usuário for 2, um ponteiro para a função `descending` é passado para a função `selectionSort` (linha 45), fazendo com que o array seja classificado em ordem decrescente.

O parâmetro a seguir aparece na linha 60 do cabeçalho de função de `selectionSort`:

```
bool ( *compare )( int, int )
```

Esse parâmetro especifica um ponteiro para uma função. A palavra-chave `bool` indica que a função sendo apontada retorna um valor `bool`. O texto `( *compare )` indica o nome do ponteiro para a função (o `*` indica que o parâmetro `compare` é um ponteiro). O texto `( int, int )` indica que a função apontada por comparação aceita dois argumentos do tipo inteiro. Os parênteses são necessários em torno de `*compare` para indicar que `compare` é um ponteiro para uma função. Se não tivéssemos incluído os parênteses, a declaração teria sido

```
bool *compare( int, int )
```

que declara uma função que recebe dois inteiros como parâmetros e retorna um ponteiro para um valor `bool`.

O parâmetro correspondente no protótipo de função de `selectionSort` é

```
bool (*)( int, int )
```

Observe que só foram incluídos tipos. Como sempre, para propósitos de documentação, o programador pode incluir nomes que o compilador irá ignorar.

A função passada para `selectionSort` é chamada na linha 71 como mostrado a seguir:

```
( *compare )( work[ smallestOrLargest ], work[ index ] )
```

Assim como um ponteiro para uma variável é desreferenciado para acessar o valor da variável, um ponteiro para uma função é desreferenciado para executar a função. Os parênteses em torno de `*compare` são novamente necessários — se não fossem incluídos, o operador `*` tentaria desreferenciar o valor retornado a partir da chamada de função. A chamada à função poderia ter sido feita sem desreferenciar o ponteiro, como em

```
compare( work[ smallestOrLargest ], work[ index ] )
```

que utiliza o ponteiro diretamente como o nome de função. Preferimos o primeiro método de chamar uma função por um ponteiro, porque ilustra explicitamente que `compare` é um ponteiro para uma função que é desreferenciado para chamar a função. O segundo método de chamar uma função por um ponteiro faz parecer que `compare` seria o nome de uma função real no programa. Isso pode ser confuso para um usuário do programa que gostaria de ver a definição de função `compare` e descobre que ela não está definida no arquivo.

```

1 // Figura 8.28: fig08_28.cpp
2 // Programa de classificação para múltiplos propósitos usando ponteiros de função.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 #include <iomanip>
9 using std::setw;
10
11 // protótipos
12 void selectionSort( int [], const int, bool (*)( int, int ) );
13 void swap( int * const, int * const );
14 bool ascending( int, int ); // implementa ordem crescente
15 bool descending( int, int ); // implementa ordem decrescente
16
17 int main()
18 {
19     const int arraySize = 10;
20     int order; // 1 = crescente, 2 = decrescente
21     int counter; // índice do array
22     int a[ arraySize ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
23
24     cout << "Enter 1 to sort in ascending order,\n"
25          << "Enter 2 to sort in descending order: ";
26     cin >> order;
27     cout << "\nData items in original order\n";
28
29     // gera saída do array original
30     for ( counter = 0; counter < arraySize; counter++ )
31         cout << setw( 4 ) << a[ counter ];
32
33     // classifica o array em ordem crescente; passa a função ascending
34     // como um argumento para especificar a ordem de classificação ascendente
35     if ( order == 1 )
36     {
37         selectionSort( a, arraySize, ascending );
38         cout << "\nData items in ascending order\n";
39     } // fim do if
40
41     // classifica o array em ordem decrescente; passa a função descending
42     // como um argumento para especificar a ordem de classificação decrescente
43     else
44     {
45         selectionSort( a, arraySize, descending );
46         cout << "\nData items in descending order\n";
47     } // fim da parte else do if...else
48

```

Figura 8.28 Programa de classificação para múltiplos propósitos utilizando ponteiros de função.

(continua)

```

49 // gera a saída do array classificado
50 for ( counter = 0; counter < arraySize; counter++ )
51     cout << setw( 4 ) << a[ counter ];
52
53     cout << endl;
54     return 0; // indica terminação bem-sucedida
55 } // fim de main
56
57 // classificação por seleção para múltiplos propósitos; o parâmetro compare é um ponteiro para
58 // a função compare que determina a ordem de classificação
59 void selectionSort( int work[], const int size,
60                    bool (*compare)( int, int ) )
61 {
62     int smallestOrLargest; // índice do menor (ou maior) elemento
63
64     // itera sobre size - 1 elementos
65     for ( int i = 0; i < size - 1; i++ )
66     {
67         smallestOrLargest = i; // primeiro índice do vetor restante
68
69         // itera para localizar o índice do menor (ou maior) elemento
70         for ( int index = i + 1; index < size; index++ )
71             if ( !(*compare)( work[ smallestOrLargest ], work[ index ] ) )
72                 smallestOrLargest = index;
73
74         swap( &work[ smallestOrLargest ], &work[ i ] );
75     } // fim do if
76 } // fim da função selectionSort
77
78 // troca os valores nas posições da memória para as quais
79 // element1Ptr e element2Ptr apontem
80 void swap( int * const element1Ptr, int * const element2Ptr )
81 {
82     int hold = *element1Ptr;
83     *element1Ptr = *element2Ptr;
84     *element2Ptr = hold;
85 } // fim da função swap
86
87 // determina se o elemento a é menor que o
88 // elemento b para uma classificação em ordem crescente
89 bool ascending( int a, int b )
90 {
91     return a < b; // retorna true se a for menor que b
92 } // fim da função ascending
93
94 // determina se o elemento a é maior que o
95 // elemento b para uma classificação em ordem decrescente
96 bool descending( int a, int b )
97 {
98     return a > b; // retorna true se a for maior que b
99 } // fim da função descending

```

Figura 8.28 Programa de classificação para múltiplos propósitos utilizando ponteiros de função.

(continua)

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1
```

```
Data items in original order
 2  6  4  8 10 12 89 68 45 37
Data items in ascending order
 2  4  6  8 10 12 37 45 68 89
```

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 2
```

```
Data items in original order
 2  6  4  8 10 12 89 68 45 37
Data items in descending order
89 68 45 37 12 10  8  6  4  2
```

Figura 8.28 Programa de classificação para múltiplos propósitos utilizando ponteiros de função.

(continuação)

Arrays de ponteiros para funções

Uma utilização de ponteiros de função é em sistemas baseados em menus. Por exemplo, um programa poderia pedir para um usuário selecionar uma opção de um menu inserindo valores de um inteiro. A escolha do usuário pode ser utilizada como um subscrito para um array de ponteiros de função e o ponteiro no array pode ser utilizado para chamar a função.

A Figura 8.29 fornece um exemplo mecânico que demonstra como declarar e utilizar um array de ponteiros para funções. O programa define três funções — `function0`, `function1` e `function2` — cada uma aceita um argumento de inteiro e não retorna um valor. A linha 17 armazena ponteiros para essas três funções no array `f`. Nesse caso, todas as funções para as quais o array aponta devem ter o mesmo tipo de retorno e os mesmos tipos de parâmetro. A declaração na linha 17 é lida começando no conjunto de parênteses mais à esquerda como, ‘`f` é um array de três ponteiros para funções que aceitam um `int` como um argumento e retornam `void`’. O array é inicializado com os nomes das três funções (que, novamente, são ponteiros). O programa pede para o usuário inserir um número entre 0 e 2 ou 3 para terminar. Quando o usuário insere um valor entre 0 e 2, o valor é utilizado como o subscrito no array de ponteiros para funções. A linha 29 invoca uma das funções no array `f`. Na chamada, `f[ choice ]` seleciona o ponteiro na localização `choice` do array. O ponteiro é desreferenciado para chamar a função e `choice` é passado como o argumento à função. Cada função imprime o valor do seu argumento e seu nome de função para indicar que a função foi chamada corretamente. Nos exercícios, você desenvolverá um sistema baseado em menus. Veremos no Capítulo 13, “Programação orientada a objetos: polimorfismo”, que os arrays de ponteiros para funções são utilizados por desenvolvedores de compiladores para implementar os mecanismos que suportam as funções `virtual` — a tecnologia-chave por trás do polimorfismo.

```
1 // Figura 8.29: fig08_29.cpp
2 // Demonstrando um array de ponteiros para funções.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 // protótipos de função -- cada função realiza ações semelhantes
9 void function0( int );
10 void function1( int );
11 void function2( int );
12
13 int main()
14 {
15     // inicializa array de 3 ponteiros para funções que
16     // aceitam um argumento int e retornam void
```

Figura 8.29 Array de ponteiros para funções.

(continua)


```

17 void (*f[ 3 ])( int ) = { function0, function1, function2 };
18
19 int choice;
20
21 cout << "Enter a number between 0 and 2, 3 to end: ";
22 cin >> choice;
23
24 // processa escolha do usuário
25 while ( ( choice >= 0 ) && ( choice < 3 ) )
26 {
27     // invoca a função na localização choice no
28     // array f e passa choice como um argumento
29     (*f[ choice ])( choice );
30
31     cout << "Enter a number between 0 and 2, 3 to end: ";
32     cin >> choice;
33 } // fim do while
34
35 cout << "Program execution completed." << endl;
36 return 0; // indica terminação bem-sucedida
37 } // fim de main
38
39 void function0( int a )
40 {
41     cout << "You entered " << a << " so function0 was called\n\n";
42 } // fim da função function0
43
44 void function1( int b )
45 {
46     cout << "You entered " << b << " so function1 was called\n\n";
47 } // fim da função function1
48
49 void function2( int c )
50 {
51     cout << "You entered " << c << " so function2 was called\n\n";
52 } // fim da função function2

```

```

Enter a number between 0 and 2, 3 to end: 0
You entered 0 so function0 was called

```

```

Enter a number between 0 and 2, 3 to end: 1
You entered 1 so function1 was called

```

```

Enter a number between 0 and 2, 3 to end: 2
You entered 2 so function2 was called

```

```

Enter a number between 0 and 2, 3 to end: 3
Program execution completed.

```

Figura 8.29 Array de ponteiros para funções.

(continuação)

8.13 Introdução ao processamento de string baseada em ponteiro

Nesta seção, introduzimos algumas funções comuns da C++ Standard Library que facilitam o processamento de string. As técnicas aqui discutidas são apropriadas para desenvolver editores de texto, processadores de texto, software de layout de página, sistemas computadorizados de composição e outros tipos de software de processamento de textos. Já utilizamos a classe `string` da C++ Standard Library em vários exemplos para representar strings como objetos completos com todos os recursos. Por exemplo, o estudo de caso da classe

GradeBook nos capítulos 3–7 representa um nome de curso que usa um objeto `string`. No Capítulo 18 apresentamos a classe `string` em detalhes. Embora utilizar os objetos `string` seja normalmente simples e direto, nesta seção, utilizamos strings baseadas em ponteiro terminadas por caractere nulo. Muitas funções da C++ Standard Library operam apenas em strings baseadas em ponteiro terminadas por caractere nulo, que são mais complicadas de utilizar do que os objetos `string`. Além disso, se você trabalha com programas C++ legados, pode ser solicitado a manipular essas strings baseadas em ponteiro.

8.13.1 Fundamentos de caracteres e strings baseadas em ponteiro

Os caracteres são os blocos de construção fundamentais do código-fonte dos programas C++. Cada programa é composto de uma sequência de caracteres que — quando agrupados entre si significativamente — é interpretada pelo compilador como uma série de instruções utilizadas para realizar uma tarefa. Um programa pode conter **caracteres constantes**. Um caractere constante é um valor de inteiro representado como caractere entre aspas simples. O valor de uma constante de caractere é o valor inteiro do caractere no conjunto de caracteres da máquina. Por exemplo, `'z'` representa o valor inteiro de `z` (122 no conjunto de caracteres ASCII; ver Apêndice B) e `'\n'` representa o valor inteiro de nova linha (10 no conjunto de caracteres ASCII).

Uma string é uma série de caracteres tratada como uma unidade única. Uma string pode incluir letras, dígitos e vários **caracteres especiais** como `+`, `-`, `*`, `/` e `$`. Os **literais string**, ou **constantes string**, em C++ são escritos em aspas duplas como mostrado a seguir:

```
"John Q. Doe"           (um nome)
"9999 Main Street"      (um endereço)
"Maynard, Massachusetts" (uma cidade e um estado)
"(201) 555-1212"        (um número de telefone)
```

Uma string baseada em ponteiro em C++ é um array de caracteres que acabam no caractere nulo (`'\0'`), que marca onde a string termina na memória. Uma string é acessada via um ponteiro para seu primeiro caractere. O valor de uma string é o endereço de seu primeiro caractere. Portanto, em C++, é apropriado dizer que *uma string é um ponteiro constante* — de fato, um ponteiro para o primeiro caractere da string. Nesse sentido, as strings são como arrays, porque um nome de array é também um ponteiro para seu primeiro elemento.

Um literal string pode ser utilizado como um inicializador na declaração de um array de caracteres ou de uma variável de tipo `char *`. Cada uma das declarações

```
char color[] = "blue";
const char *colorPtr = "blue";
```

inicializa uma variável como a string `"blue"`. A primeira declaração cria um array `color` de cinco elementos contendo o caractere `'b'`, `'l'`, `'u'`, `'e'` e `'\0'`. A segunda declaração cria o ponteiro de variável `colorPtr` que aponta para a letra `b` na string `"blue"` (que acaba em `'\0'`) em algum lugar da memória. Os literais string têm a classe de armazenamento `static` (eles existem até o fim do programa) e podem ou não ser compartilhados se o mesmo literal string for referenciado a partir de múltiplas localizações em um programa. Além disso, os literais string em C++ são constantes — seus caracteres não podem ser modificados.

A declaração `char color[] = "blue";` também poderia ser escrita

```
char color[] = { 'b', 'l', 'u', 'e', '\0' };
```

Ao declarar um array de caractere para conter uma string, o array deve ser suficientemente grande para armazenar a string e seu caractere de terminação nulo. A declaração anterior determina o tamanho do array, com base no número de inicializadores fornecidos na lista inicializadora.



Erro comum de programação 8.15

É um erro não alocar espaço suficiente em um array de caracteres para armazenar o caractere nulo que termina uma string.



Erro comum de programação 8.16

Criar ou utilizar uma string no estilo C que não contém um caractere nulo de terminação pode levar a erros de lógica.



Dica de prevenção de erro 8.4

Ao armazenar uma string de caracteres em um array de caracteres, certifique-se de que o array é grande o bastante para armazenar a maior string que será armazenada. O C++ permite que strings de qualquer comprimento sejam armazenadas. Se uma string for mais longa que o array de caracteres em que ela deve ser armazenada, os caracteres além do fim do array sobrescreverão os dados na memória seguinte ao array, resultando em erros de lógica.

Uma string pode ser lida em um array de caracteres utilizando a extração de fluxo com `cin`. Por exemplo, a seguinte instrução pode ser utilizada a fim de ler uma string para o array de caracteres `word[ 20 ]`:

```
cin >> word;
```

A string inserida pelo usuário é armazenada em `word`. A instrução precedente lê caracteres até que um caractere de espaço em branco ou indicador de fim do arquivo seja encontrado. Observe que a string não deve ter mais que 19 caracteres para deixar espaço para o

caractere de terminação nulo. O manipulador de fluxo `setw` pode ser utilizado para assegurar que a string lida em `word` não exceda o tamanho do array. Por exemplo, a instrução

```
cin >> setw( 20 ) >> word;
```

especifica que `cin` deve ler um máximo de 19 caracteres no array `word` e salvar a 20ª posição no array para armazenar o caractere de terminação nulo para a string. O manipulador de fluxo `setw` só se aplica ao próximo valor sendo inserido. Se mais de 19 caracteres forem inseridos, os caracteres restantes não são salvos em `word`, mas serão lidos e podem ser armazenados em outra variável.

Em alguns casos, é desejável inserir uma linha inteira de texto em um array. Para esse propósito, o C++ fornece a função `cin.getline` no arquivo de cabeçalho `<iostream>`. No Capítulo 3 você foi apresentado à função `getline` similar do arquivo de cabeçalho `<string>`, que lia a entrada até que um caractere de nova linha fosse inserido e armazenava a entrada (sem o caractere de nova linha) em uma string especificada como um argumento. A função `cin.getline` aceita três argumentos — um array de caracteres em que a linha de texto será armazenada, um comprimento e um caractere delimitador. Por exemplo, o segmento de programa

```
char sentence[ 80 ];
cin.getline( sentence, 80, '\n' );
```

declara o array `sentence` de 80 caracteres e lê uma linha de texto do teclado no array. A função pára de ler caracteres quando o caractere delimitador `'\n'` é encontrado, quando o indicador de fim do arquivo é inserido ou quando o número de caracteres lidos até agora é um menor que o comprimento especificado no segundo argumento. (O último caractere no array é reservado para o caractere de terminação nulo.) Se o caractere delimitador é encontrado, ele é lido e descartado. O terceiro argumento para `cin.getline` tem `'\n'` como um valor-padrão, então a chamada de função precedente poderia ter sido escrita como mostrado a seguir:

```
cin.getline( sentence, 80 );
```

O Capítulo 15, “Entrada/saída de fluxo”, fornece uma discussão detalhada sobre `cin.getline` e outras funções de entrada/saída.



Erro comum de programação 8.17

Processar um único caractere como uma string `char *` pode levar a um erro fatal de tempo de execução. Uma string `char *` é um ponteiro — provavelmente um inteiro bem grande. Entretanto, um caractere é um inteiro pequeno (valores ASCII variam de 0–255). Em muitos sistemas, desreferenciar um valor `char` causa um erro, porque os endereços de memória baixa são reservados para usos especiais como handlers de interrupção de sistemas operacionais — portanto, ocorrem as ‘violações de acesso de memória’.



Erro comum de programação 8.18

É um erro de compilação passar uma string como um argumento para uma função quando um caractere é esperado.

8.13.2 Funções de manipulação de string da biblioteca de tratamento de strings

A biblioteca de tratamento de strings fornece muitas funções úteis para manipular dados de string, comparar strings, pesquisar por caracteres e outras strings em strings, tokenizar strings (separá-las em partes lógicas como as palavras separadas em uma frase) e determinar seu comprimento. Esta seção apresenta algumas funções de manipulação de strings comuns da biblioteca de tratamento de strings (da biblioteca-padrão C++). As funções são resumidas na Figura 8.30; em seguida, cada uma delas é utilizada em um exemplo de ‘código ativo’ (*live-code*). Os protótipos para essas funções são encontrados no arquivo de cabeçalho `<cstring>`.

Observe que várias funções na Figura 8.30 contêm parâmetros com tipo de dados `size_t`. Esse tipo é definido no arquivo de cabeçalho `<cstring>` como um tipo integral sem sinal como `unsigned int` ou `unsigned long`.



Erro comum de programação 8.19

Esquecer de incluir o arquivo de cabeçalho `<cstring>` ao usar funções da biblioteca de tratamento de strings causa erros de compilação.

Copiando strings com `strcpy` e `strncpy`

A função `strcpy` copia seu segundo argumento — uma string — para seu primeiro argumento — um array de caracteres que deve ser grande o bastante para armazenar a string e seu caractere de terminação nulo (que também é copiado). A função `strncpy` é equivalente à `strcpy`, exceto pelo fato de que `strncpy` especifica o número de caracteres a serem copiados da string para o array. Observe que a função `strncpy` não copia necessariamente o caractere de terminação nulo de seu segundo argumento — um caractere de terminação nulo só é escrito se o número de caracteres a ser copiado for pelo menos um maior que o comprimento da string. Por exemplo, se `"test"` é o segundo argumento, um caractere de terminação nulo só é escrito se o terceiro argumento para `strncpy` for de pelo menos 5 (quatro caracteres em `"test"` mais um caractere de terminação nulo). Se o terceiro argumento for maior que 5, os caracteres nulos serão acrescentados ao array até que o número total de caracteres especificado pelo terceiro argumento seja escrito.

Protótipo da função	Descrição da função
<code>char *strcpy(char *s1, const char *s2);</code>	Copia a string s2 para o array de caracteres s1. O valor de s1 é retornado.
<code>char *strncpy(char *s1, const char *s2, size_t n);</code>	Copia no máximo n caracteres da string s2 para o array de caractere s1. O valor de s1 é retornado.
<code>char *strcat(char *s1, const char *s2);</code>	Acrescenta a string s2 a s1. O primeiro caractere de s2 sobrescreve o caractere de terminação nulo de s1. O valor de s1 é retornado.
<code>char *strncat(char *s1, const char *s2, size_t n);</code>	Acrescenta no máximo n caracteres da string s2 à string s1. O primeiro caractere de s2 sobrescreve o caractere de terminação nulo de s1. O valor de s1 é retornado.
<code>int strcmp(const char *s1, const char *s2);</code>	Compara a string s1 com a string s2. A função retorna um valor zero, menor que zero (normalmente -1) ou maior que zero (normalmente 1) se s1 for igual, maior ou menor que s2, respectivamente.
<code>int strncmp(const char *s1, const char *s2, size_t n);</code>	Compara até n caracteres da string s1 com a string s2. A função retorna zero, menor que zero ou maior que zero se a parte de n caracteres de s1 é igual a, maior ou menor que a parte correspondente de n caracteres de s2, respectivamente.
<code>char *strtok(char *s1, const char *s2);</code>	Uma sequência de chamadas a strtok quebra a string s1 em 'tokens' — partes lógicas como as palavras em uma linha de texto. A string é dividida com base nos caracteres contidos na string s2. Por exemplo, se fôssemos quebrar a string "this:is:a:string" em tokens com base no caractere ': ', os tokens resultantes seriam "this", "is", "a" e "string". Entretanto, a função strtok retorna somente um token por vez. A primeira chamada contém s1 como o primeiro argumento, e as chamadas subsequentes que continuam tokenizando a mesma string contêm NULL como o primeiro argumento. Cada chamada retorna um ponteiro para o token atual. Se não houver mais tokens quando a função for chamada, NULL é retornado.
<code>size_t strlen(const char *s);</code>	Determina o comprimento da string s. O número de caracteres que precedem o caractere de terminação nulo é retornado.

Figura 8.30 Funções de manipulação de string da biblioteca de tratamento de strings.



Erro comum de programação 8.20

*Ao utilizar strncpy, o caractere de terminação nulo do segundo argumento (uma string char *) não será copiado se o número de caracteres especificado pelo terceiro argumento de strncpy não for maior que o comprimento do segundo argumento. Nesse caso, pode ocorrer um erro fatal se o programador não terminar manualmente a string char * resultante com um caractere nulo.*

A Figura 8.31 utiliza strcpy (linha 17) para copiar a string inteira do array x para o array y, e utiliza strncpy (linha 23) para copiar os primeiros 14 caracteres do array x para o array z. A linha 24 acrescenta um caractere nulo ('\0') ao array z, porque a chamada a strncpy no programa não escreve um caractere de terminação nulo. (O terceiro argumento é menor que o comprimento de string do segundo argumento mais um.)

Concatenando strings com strcat e strncat

A função **strcat** acrescenta seu segundo argumento (uma string) a seu primeiro argumento (um array de caracteres contendo uma string). O primeiro caractere do segundo argumento substitui o caractere nulo ('\0') que termina a string no primeiro argumento. O programador deve assegurar que o array utilizado para armazenar a primeira string é suficientemente grande para armazenar a combinação da primeira e da segunda strings e do caractere de terminação nulo (copiado a partir da segunda string). A função **strncat** acrescenta um número especificado de caracteres da segunda string para a primeira string e acrescenta um caractere de terminação nulo ao resultado. O programa da Figura 8.32 demonstra a função **strcat** (linhas 19 e 29) e a função **strncat** (linha 24).

```

1 // Figura 8.31: fig08_31.cpp
2 // Utilizando strcpy e strncpy.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // protótipos para strcpy e strncpy
8 using std::strcpy;
9 using std::strncpy;
10
11 int main()
12 {
13     char x[] = "Happy Birthday to You"; // o comprimento da string é 21
14     char y[ 25 ];
15     char z[ 15 ];
16
17     strcpy( y, x ); // copia conteúdo de x para y
18
19     cout << "The string in array x is: " << x
20         << "\nThe string in array y is: " << y << '\n';
21
22     // copia os primeiros 14 caracteres de x para z
23     strncpy( z, x, 14 ); // não copia o caractere nulo
24     z[ 14 ] = '\0'; // acrescenta '\0' ao conteúdo de z
25
26     cout << "The string in array z is: " << z << endl;
27     return 0; // indica terminação bem-sucedida
28 } // fim de main

```

The string in array x is: Happy Birthday to You
The string in array y is: Happy Birthday to You
The string in array z is: Happy Birthday

Figura 8.31 strcpy e strncpy.

```

1 // Figura 8.32: fig08_32.cpp
2 // Utilizando strcat e strncat.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // protótipos para strcat e strncat
8 using std::strcat;
9 using std::strncat;
10
11 int main()
12 {
13     char s1[ 20 ] = "Happy "; // comprimento 6
14     char s2[] = "New Year "; // comprimento 9
15     char s3[ 40 ] = "";
16
17     cout << "s1 = " << s1 << "\ns2 = " << s2;
18
19     strcat( s1, s2 ); // concatena s2 com s1 (comprimento 15)

```

Figura 8.32 strcat e strncat.

(continua)

```

20
21     cout << "\n\nAfter strcat(s1, s2):\ns1 = " << s1 << "\ns2 = " << s2;
22
23     // concatena os 6 primeiros caracteres de s1 a s3
24     strncat( s3, s1, 6 ); // coloca '\0' depois de último caractere
25
26     cout << "\n\nAfter strncat(s3, s1, 6):\ns1 = " << s1
27         << "\ns3 = " << s3;
28
29     strcat( s3, s1 ); // concatena s1 a s3
30     cout << "\n\nAfter strcat(s3, s1):\ns1 = " << s1
31         << "\ns3 = " << s3 << endl;
32     return 0; // indica terminação bem-sucedida
33 } // fim de main

```

```

s1 = Happy
s2 = New Year

```

```

After strcat(s1, s2):
s1 = Happy New Year
s2 = New Year

```

```

After strncat(s3, s1, 6):
s1 = Happy New Year
s3 = Happy

```

```

After strcat(s3, s1):
s1 = Happy New Year
s3 = Happy Happy New Year

```

Figura 8.32 strcat e strncat.

(continuação)

Comparando strings com **strcmp** e **strncmp**

A Figura 8.33 compara três strings utilizando **strcmp** (linhas 21, 22 e 23) e **strncmp** (linhas 26, 27 e 28). A função **strcmp** compara, caractere por caractere, seu primeiro argumento de string com seu segundo argumento de string. A função retornará zero se as strings forem iguais, um valor negativo se a primeira string for menor que a segunda string e um valor positivo se a primeira string for maior que a segunda. A função **strncmp** é equivalente à **strcmp**, exceto pelo fato de que **strncmp** compara até um número especificado de caracteres. A função **strncmp** pára de comparar caracteres se alcançar o caractere nulo em um de seus argumentos de string. O programa imprime o valor inteiro retornado em cada chamada de função.



Erro comum de programação 8.21

*Pressupor que **strcmp** e **strncmp** retornam 1 (valor verdadeiro) quando seus argumentos são iguais é um erro de lógica. Ambas as funções retornam zero (valor falso do C++) para igualdade. Portanto, ao testar a igualdade de duas strings, o resultado da função **strcmp** ou **strncmp** deve ser comparado com zero para determinar se as strings são iguais.*

Para entender exatamente o que significa uma string ser ‘maior que’ ou ‘menor que’ outra string, considere o processo de alfabetar uma série de sobrenomes. O leitor iria, sem dúvida, colocar ‘Jones’ antes de ‘Smith’ porque a primeira letra de ‘Jones’ vem antes da primeira letra de ‘Smith’ no alfabeto. Mas o alfabeto é mais que uma simples lista de 26 letras — é uma lista ordenada de caracteres. Cada letra ocorre em uma posição específica dentro da lista. O ‘z’ é mais que apenas uma letra do alfabeto; ‘z’ é especificamente a 26ª letra do alfabeto.

Como o computador sabe que uma letra vem antes de outra? Todos os caracteres são representados dentro do computador como códigos numéricos; quando o computador compara duas strings, na realidade, ele compara os códigos numéricos dos caracteres nas strings.

Em um esforço de padronizar representações de caractere, a maioria dos fabricantes de computador projetou suas máquinas para utilizar um dos dois esquemas de codificação populares — o ASCII ou **EBCDIC**. Lembre-se de que ASCII significa ‘American Standard Code for Information Interchange’. EBCDIC quer dizer ‘Extended Binary Coded Decimal Interchange Code’. Existem outros esquemas de codificação, mas esses dois são os mais populares.

O ASCII e o EBCDIC são chamados de **códigos de caractere** ou conjuntos de caracteres. A maioria dos leitores deste livro utilizará computadores desktop ou notebooks que empregam o conjunto de caracteres ASCII. Os computadores mainframe da IBM utilizam o conjunto de caracteres EBCDIC. Como o uso da Internet e da World Wide Web torna-se intenso no mundo todo, o mais novo conjunto de caracteres Unicode está crescendo rapidamente em popularidade. Para informações adicionais sobre o Unicode, visite www.unicode.org. As manipulações de strings e caracteres, na realidade, envolvem a manipulação dos códigos numéricos apropriados e não os caracteres em si. Isso explica a intercambialidade entre caracteres e inteiros pequenos no C++. Visto que é significativo dizer que um código numérico é maior que, menor que ou igual a outro código numérico, torna-se possível relacionar vários caracteres ou strings entre si para se referir aos códigos de caractere. O Apêndice B contém os códigos de caractere ASCII.



Dica de portabilidade 8.5

Os códigos numéricos internos utilizados para representar caracteres podem ser diferentes em computadores diferentes, porque esses computadores podem utilizar conjuntos de caracteres diferentes.



Dica de portabilidade 8.6

Não teste explicitamente os códigos ASCII, como em `if ( rating == 65 );` em vez disso, use a constante de caractere correspondente, como em `if ( rating == 'A' );`.

[Nota: Com alguns compiladores, as funções `strcmp` e `strncmp` sempre retornam -1, 0 ou 1, como na saída de exemplo da Figura 8.33. Com outros compiladores, essas funções retornam 0 ou a diferença entre os códigos numéricos dos primeiros caracteres que diferem nas strings sendo comparadas. Por exemplo, quando `s1` e `s3` são comparados, os primeiros caracteres que diferem entre eles são os primeiros caracteres da segunda palavra em cada string — N (código numérico 78) em `s1` e H (código numérico 72) em `s3`, respectivamente. Nesse caso, o valor de retorno será 6 (ou -6 se `s3` for comparado com `s1`).]

```

1 // Figura 8.33: fig08_33.cpp
2 // Utilizando strcmp e strncmp.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 #include <cstring> // protótipos para strcmp e strncmp
11 using std::strcmp;
12 using std::strncmp;
13
14 int main()
15 {
16     char *s1 = "Happy New Year";
17     char *s2 = "Happy New Year";
18     char *s3 = "Happy Holidays";
19
20     cout << "s1 = " << s1 << "\ns2 = " << s2 << "\ns3 = " << s3
21         << "\n\nstrcmp(s1, s2) = " << setw( 2 ) << strcmp( s1, s2 )
22         << "\nstrcmp(s1, s3) = " << setw( 2 ) << strcmp( s1, s3 )
23         << "\nstrcmp(s3, s1) = " << setw( 2 ) << strcmp( s3, s1 );
24
25     cout << "\n\nstrncmp(s1, s3, 6) = " << setw( 2 )
26         << strncmp( s1, s3, 6 ) << "\nstrncmp(s1, s3, 7) = " << setw( 2 )
27         << strncmp( s1, s3, 7 ) << "\nstrncmp(s3, s1, 7) = " << setw( 2 )
28         << strncmp( s3, s1, 7 ) << endl;
29     return 0; // indica terminação bem-sucedida
30 } // fim de main

```

Figura 8.33 strcmp e strncmp.

(continua)


```

s1 = Happy New Year
s2 = Happy New Year
s3 = Happy Holidays

strcmp(s1, s2) = 0
strcmp(s1, s3) = 1
strcmp(s3, s1) = -1

strncmp(s1, s3, 6) = 0
strncmp(s1, s3, 7) = 1
strncmp(s3, s1, 7) = -1

```

Figura 8.33 strcmp e strncmp.

(continuação)

Tokenizando uma string com *strtok*

A função **strtok** divide uma string em uma série de **tokens**. Token é uma sequência de caracteres separados por **caracteres delimitadores** (normalmente espaços ou sinais de pontuação). Por exemplo, em uma linha de texto, cada palavra pode ser considerada um token, e os espaços que separam as palavras podem ser considerados delimitadores.

São necessárias múltiplas chamadas para **strtok** para se dividir uma string em tokens (supondo que a string contenha mais de um token). A primeira chamada a **strtok** contém dois argumentos, uma string a ser tokenizada e uma string contendo caracteres que separam os tokens (isto é, delimitadores). A linha 19 na Figura 8.34 atribui a **tokenPtr** um ponteiro para o primeiro token em **sentence**. O segundo argumento, " ", indica que os tokens em **sentence** são separados por espaços. A função **strtok** procura o primeiro caractere na **sentence** que não seja um caractere delimitador (espaço). Isso no primeiro token. A função então localiza o próximo caractere delimitador na string e o substitui por um caractere nulo ('\0'). Isso termina o token atual. A função **strtok** salva (em uma variável **static**) um ponteiro para o próximo caractere que se segue ao token em **sentence** e retorna um ponteiro para o token atual.

As chamadas subsequentes a **strtok** para continuar tokenizando **sentence** contêm **NULL** como o primeiro argumento (linha 25). O argumento **NULL** indica que a chamada a **strtok** deve continuar tokenizando a partir da localização em **sentence** salva pela última chamada a **strtok**. Observe que **strtok** mantém essas informações salvas de modo que não fiquem visíveis ao programador. Se nenhum token restar quando **strtok** for chamado, **strtok** retornará **NULL**. O programa da Figura 8.34 utiliza **strtok** para tokenizar a string "This is a sentence with 7 tokens". O programa imprime cada token em uma linha separada. A linha 28 gera saída de **sentence** depois da tokenização. Observe que *strtok modifica a string de entrada*; portanto, uma cópia da string deve ser feita se o programa exigir a original depois das chamadas para **strtok**. Quando **sentence** é enviada para a saída depois da tokenização, observe que somente a palavra 'This' é impressa, porque **strtok** substituiu cada espaço em branco em **sentence** por um caractere nulo ('\0') durante o processo de tokenização.

```

1 // Figura 8.34: fig08_34.cpp
2 // Utilizando strtok.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // protótipo para strtok
8 using std::strtok;
9
10 int main()
11 {
12     char sentence[] = "This is a sentence with 7 tokens";
13     char *tokenPtr;
14
15     cout << "The string to be tokenized is:\n" << sentence
16         << "\n\nThe tokens are:\n\n";
17

```

Figura 8.34 strtok.

(continua)

```

18 // inicia a tokenização da frase
19 tokenPtr = strtok( sentence, " " );
20
21 // continua tokenizando a frase até tokenPtr tornar-se NULL
22 while ( tokenPtr != NULL )
23 {
24     cout << tokenPtr << '\n';
25     tokenPtr = strtok( NULL, " " ); // obtém o próximo token
26 } // fim do while
27
28 cout << "\nAfter strtok, sentence = " << sentence << endl;
29 return 0; // indica terminação bem-sucedida
30 } // fim de main

```

The string to be tokenized is:
This is a sentence with 7 tokens

The tokens are:

This
is
a
sentence
with
7
tokens

After strtok, sentence = This

Figura 8.34 strtok.

(continuação)



Erro comum de programação 8.22

Não perceber que strtok modifica a string tokenizada e, então, tentar utilizar essa string como se ela fosse a string original não modificada é um erro de lógica.

Determinando comprimentos de string

A função **strlen** aceita uma string como um argumento e retorna o número de caracteres na string — o caractere de terminação nulo não é incluído no comprimento. O comprimento também é o índice do caractere nulo. O programa da Figura 8.35 demonstra a função **strlen**.

```

1 // Figura 8.35: fig08_35.cpp
2 // Utilizando strlen.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // protótipo para strlen
8 using std::strlen;
9
10 int main()
11 {

```

Figura 8.35 strlen retorna o comprimento de uma string char *.

(continua)

```

12  char *string1 = "abcdefghijklmnopqrstuvwxy";
13  char *string2 = "four";
14  char *string3 = "Boston";
15
16  cout << "The length of \"" << string1 << "\" is " << strlen( string1 )
17      << "\nThe length of \"" << string2 << "\" is " << strlen( string2 )
18      << "\nThe length of \"" << string3 << "\" is " << strlen( string3 )
19      << endl;
20  return 0; // indica terminação bem-sucedida
21 } // fim de main

```

```

The length of "abcdefghijklmnopqrstuvwxy" is 26
The length of "four" is 4
The length of "Boston" is 6

```

Figura 8.35 strlen retorna o comprimento de uma string char *.

(continuação)

8.14 Síntese

Neste capítulo fornecemos uma introdução detalhada sobre ponteiros ou variáveis que contêm endereços de memória como seus valores. Começamos demonstrando como declarar e inicializar ponteiros. Você viu como utilizar o operador de endereço (&) para atribuir o endereço de uma variável a um ponteiro e o operador de indireção (*) para acessar os dados armazenados na variável indiretamente referenciada por um ponteiro. Discutimos como passar argumentos por referência utilizando tanto os argumentos de ponteiro como os argumentos de referência.

Você aprendeu a utilizar const com ponteiros para impor o princípio do menor privilégio. Demonstramos como utilizar ponteiros não constantes para dados não constantes, ponteiros não constantes para dados constantes, ponteiros constantes para dados não constantes e ponteiros constantes para dados constantes. Em seguida, utilizamos a classificação por seleção para demonstrar como passar arrays e elementos do array individuais por referência. Discutimos o operador sizeof, que pode ser utilizado para determinar o tamanho, em bytes, de um tipo de dados durante a compilação de programa.

Continuamos demonstrando como utilizar os ponteiros em expressões aritméticas e de comparação. Você viu que a aritmética de ponteiros pode ser utilizada para pular de um elemento de um array para outro. Você aprendeu a utilizar arrays de ponteiros e, mais especificamente, arrays de string (uma string de arrays). Então continuamos discutindo os ponteiros de função, que permitem aos programadores passar funções como parâmetros. Concluímos o capítulo com uma discussão sobre as várias funções C++ que manipulam strings baseadas em ponteiro. Você aprendeu as capacidades de processamento de string, por exemplo, copiar strings, tokenizá-las e determinar seu comprimento.

No próximo capítulo, iniciamos nosso tratamento aprofundado de classes. Você aprenderá sobre o escopo de membros de uma classe e sobre como manter os objetos em um estado consistente. Também aprenderá a utilizar funções-membro especiais chamadas de construtores e destrutores, que executam quando um objeto é criado e destruído, respectivamente.

Resumo

- Os ponteiros são variáveis que contêm como seus valores endereços de memória de outras variáveis.
- A declaração

```
int *ptr;
```

declara ptr como um ponteiro para uma variável do tipo int e se lê como 'ptr é um ponteiro para int'. O *, do modo usado aqui em uma declaração, indica que a variável é um ponteiro.

- Há três valores que podem ser utilizados para inicializar um ponteiro: 0, NULL ou um endereço de um objeto do mesmo tipo. A inicialização de um ponteiro como 0 e a inicialização desse mesmo ponteiro como NULL são idênticas — 0 é a convenção em C++.
- O único inteiro que pode ser atribuído a um ponteiro sem coerção é zero.
- O operador & (endereço) retorna o endereço de memória de seu operando.
- O operando do operador de endereço deve ser um nome de variável (ou outro lvalue); o operador de endereço não pode ser aplicado a constantes ou expressões que não retornam uma referência.

- O operador `*`, referido como o operador de indireção (ou desreferenciação), retorna um sinônimo, alias ou apelido para o nome do objeto para o qual seu operando aponta na memória. Isso é chamado de desreferenciar o ponteiro.
- Ao chamar uma função com um argumento que o chamador quer que a função chamada modifique, o endereço do argumento pode ser passado. A função chamada então utiliza o operador de indireção (`*`) para desreferenciar o ponteiro e modificar o valor do argumento na função chamada.
- Uma função que recebe um endereço como um argumento deve ter um ponteiro como seu parâmetro correspondente.
- O qualificador `const` permite que o programador informe ao compilador que o valor de uma variável particular não pode ser modificado pelo identificador especificado. Se uma tentativa de modificar um valor `const` for feita, o compilador emite um aviso ou um erro, dependendo do compilador particular.
- Há quatro maneiras de passar um ponteiro para uma função — um ponteiro não constante para dados não constantes, um ponteiro não constante para dados constantes, um ponteiro constante para dados não constantes e um ponteiro constante para dados constantes.
- O valor do nome do array é o endereço de (um ponteiro para) um primeiro elemento do array.
- Para passar um único elemento de um array por referência utilizando ponteiros, passe o endereço do elemento do array específico.
- O C++ fornece o operador unário `sizeof` para determinar o tamanho de um array (ou de qualquer outro tipo de dados, variável ou constante) em bytes em tempo de compilação.
- Quando aplicado ao nome de um array, o operador `sizeof` retorna o número total de bytes no array como um inteiro.
- As operações aritméticas que podem ser realizadas em ponteiros são incrementar (`++`) um ponteiro, decrementar (`--`) um ponteiro, adicionar (`+` ou `+=`) um inteiro a um ponteiro, subtrair (`-` ou `-=`) um inteiro de um ponteiro e subtrair um ponteiro de outro.
- Quando um inteiro é adicionado a ou subtraído de um ponteiro, o ponteiro é incrementado ou decrementado por esse inteiro vezes o tamanho do objeto que o ponteiro referencia.
- Dois ponteiros podem ser atribuídos reciprocamente se ambos forem do mesmo tipo. Caso contrário, uma coerção deve ser utilizada. A exceção a isso é um ponteiro `void *`, que é um tipo genérico de ponteiro que pode armazenar valores de ponteiro de qualquer tipo. Os ponteiros para `void` podem ser atribuídos a ponteiros de outros tipos. Um ponteiro `void *` pode ser atribuído a um ponteiro de outro tipo somente com uma coerção de tipo explícita.
- As únicas operações válidas em um ponteiro `void *` são a comparação de ponteiros `void *` com outros ponteiros, a atribuição de endereços a ponteiros `void *` e a coerção de ponteiros `void *` em tipos de ponteiros válidos.
- Os ponteiros podem ser comparados utilizando operadores de igualdade e relacionais. As comparações que utilizam operadores relacionais só são significativas se os ponteiros apontarem para membros do mesmo array.
- Os ponteiros que apontam para arrays podem ser subscritos exatamente como os nomes de array podem.
- Na notação de ponteiro/deslocamento, se o ponteiro apontar para o primeiro elemento do array, o deslocamento é o mesmo que um subscrito do array.
- Todas as expressões de array indexadas com subscritos podem ser escritas com um ponteiro e um deslocamento, utilizando o nome do array como um ponteiro ou um ponteiro separado que aponta para o array.
- Os arrays podem conter ponteiros.
- Um ponteiro para uma função é o endereço em que o código da função reside.
- Os ponteiros para funções podem ser passados para funções, retornados de funções, armazenados em arrays e atribuídos a outros ponteiros.
- Uma utilização comum de ponteiros de função é nos assim chamados sistemas baseados em menus. Os ponteiros de função são utilizados para selecionar qual função chamar para um item de menu particular.
- A função `strcpy` copia seu segundo argumento — uma string — para seu primeiro argumento — um array de caracteres. O programador deve assegurar que o array-alvo seja suficientemente grande para armazenar a string e seu caractere de terminação nulo.
- A função `strncpy` é equivalente à `strcpy`, exceto pelo fato de que uma chamada a `strncpy` especifica o número de caracteres a ser copiado da string para o array. O caractere de terminação nulo só será copiado se o número de caracteres a ser copiado for pelo menos um maior que o comprimento da string.
- A função `strcat` acrescenta seu segundo argumento de string — incluindo o caractere de terminação nulo — ao seu primeiro argumento de string. O primeiro caractere da segunda string substitui o caractere nulo (`'\0'`) da primeira string. O programador deve assegurar que o array-alvo utilizado para armazenar a primeira string é suficientemente grande para armazenar a primeira e a segunda strings.
- A função `strncat` é equivalente à `strcat`, exceto pelo fato de que uma chamada a `strncat` acrescenta um número especificado de caracteres da segunda string à primeira string. Um caractere de terminação nulo é acrescentado ao resultado.
- A função `strcmp` compara, caractere por caractere, seu primeiro argumento de string com seu segundo argumento de string. A função retornará zero se as strings forem iguais, um valor negativo se a primeira string for menor que a segunda string, e um valor positivo se a primeira string for maior que a segunda.

- A função `strncmp` é equivalente à `strcmp`, exceto pelo fato de que `strncmp` compara um número especificado de caracteres. Se o número de caracteres em uma das strings for menor que o número de caracteres especificado, `strncmp` compara os caracteres até o caractere nulo na string mais curta ser encontrado.
- Uma sequência de chamadas a `strtok` divide uma string em tokens que são separados por caracteres contidos em um segundo argumento de string. A primeira chamada especifica a string tokenizada como o primeiro argumento, e as chamadas subsequentes para continuar tokenizando a mesma string especificam NULL como o primeiro argumento. A função retorna um ponteiro para o token atual de cada chamada. Se não houver mais tokens quando `strtok` for chamada, NULL é retornado.
- A função `strlen` aceita uma string como um argumento e retorna o número de caracteres na string — o caractere de terminação nulo não é incluído no comprimento da string.

Terminologia

& (operador de endereço)	EBCDIC (Extended Binary Coded Decimal Interchange Code)	referenciar elementos de array
* (operador de desreferenciar ponteiro ou operador de indireção)	endereço (&), operador de função <code>getline</code> de <code>cin</code>	referenciar um valor indiretamente
'\0' (caractere nulo)	inanição	<code>size_t</code> , tipo
adiamento indefinido	incrementar um ponteiro	<code>sizeof</code> , operador
algoritmo de classificação por seleção	indireção	<code>strcat</code> , função do arquivo de cabeçalho <code><cstring></code>
argumentos da linha de comando	intercambialidade de arrays e ponteiros	<code>strcmp</code> , função do arquivo de cabeçalho <code><cstring></code>
aritmética de ponteiros	<code>islower</code> , função (<code><cctype></code>)	<code>strcpy</code> , função do arquivo de cabeçalho <code><cstring></code>
array de ponteiros para funções	modificar endereço armazenado em variável ponteiro	string sendo tokenizada
array de strings	modificar um ponteiro constante	string terminada por caractere nulo
ASCII (American Standard Code for Information Interchange)	operador de desreferência (*)	strings baseadas em ponteiro
caractere de terminação nulo	operador de desreferenciar ponteiro (*)	strings de tokenização
caractere delimitador	operador de indireção (*)	<code>strlen</code> , função do arquivo de cabeçalho <code><cstring></code>
caractere nulo (' \0 ')	passagem por referência com argumentos de ponteiro	<code>strncat</code> , função do arquivo de cabeçalho <code><cstring></code>
caracteres especiais	passagem por referência com argumentos de referência	<code>strncmp</code> , função do arquivo de cabeçalho <code><cstring></code>
chamando funções por referência	ponteiro constante	<code>strncpy</code> , função do arquivo de cabeçalho <code><cstring></code>
código de caractere	ponteiro constante para dados constantes	<code>strtok</code> , função do arquivo de cabeçalho <code><cstring></code>
comparando strings	ponteiro constante para dados não-constantes	subtração de ponteiro
concatenando strings	ponteiro de função	token
<code>const</code> com parâmetros de função	ponteiro não-constante para dados constantes	<code>toupper</code> , função (<code><cctype></code>)
constante de caractere	ponteiro não-constante para dados não-constantes	
constante string	ponteiro nulo	
cópia de string	ponteiro para uma função	
copiando strings	referência a dados constantes	
decrementar um ponteiro	referenciar diretamente um valor	
deslocamento para um ponteiro		
desreferenciar um ponteiro		
desreferenciar um ponteiro 0		

Exercícios de revisão

- 8.1** Complete cada uma das seguintes sentenças:
- Um ponteiro é uma variável que contém como seu valor o(a) _____ de outra variável.
 - Os três valores que podem ser utilizados para inicializar um ponteiro são _____, _____ e _____.
 - O único inteiro que pode ser atribuído diretamente a um ponteiro é _____.
- 8.2** Determine se as seguintes sentenças são *verdadeiras* ou *falsas*. Se a resposta for *falsa*, explicar por quê.
- O operador de endereço & pode ser aplicado somente a constantes e expressões.
 - Um ponteiro que é declarado como tipo `void *` pode ser desreferenciado.
 - Os ponteiros de tipos diferentes nunca podem ser atribuídos um ao outro sem a operação de coerção.
- 8.3** Para cada um dos itens a seguir, escreva instruções C++ que realizam a tarefa especificada. Suponha que números de dupla precisão com ponto flutuante sejam armazenados em oito bytes e que o endereço inicial do array esteja na posição 1002500 na memória. Cada parte do exercício deve utilizar os resultados de partes anteriores onde apropriado.

- Declare um array do tipo `double` chamado `numbers` com 10 elementos, e inicialize os elementos para os valores 0.0, 1.1, 2.2, ..., 9.9. Suponha que a constante simbólica `SIZE` foi definida como 10.
- Declare um ponteiro `nPtr` que aponta para uma variável do tipo `double`.
- Utilize uma instrução `for` para imprimir os elementos do array `numbers` usando notação de array subscripto. Imprima cada número com uma casa decimal de precisão à direita do ponto de fração decimal.
- Escreva duas instruções separadas que atribuem, cada uma, o endereço inicial do array `numbers` à variável ponteiro `nPtr`.
- Utilize uma instrução `for` para imprimir os elementos do array `numbers` utilizando a notação de ponteiro/deslocamento com o ponteiro `nPtr`.
- Utilize uma instrução `for` para imprimir os elementos do array `numbers` utilizando a notação de ponteiro/deslocamento com o nome do array como o ponteiro.
- Utilize uma instrução `for` para imprimir os elementos do array `numbers` utilizando a notação de ponteiro/subscrito com o ponteiro `nPtr`.
- Referencie o quarto elemento do array `numbers` utilizando a notação de subscripto de array, a notação de ponteiro/deslocamento com o nome de array como o ponteiro, a notação de subscripto de ponteiro com `nPtr` e a notação de ponteiro/deslocamento com `nPtr`.
- Supondo que `nPtr` aponta para o começo do array `numbers`, que endereço é referenciado por `nPtr + 8`? Que valor é armazenado nessa localização?
- Supondo que `nPtr` aponta para `numbers[5]`, que endereço é referenciado por `nPtr` depois de `nPtr -= 4` ser executado? Qual valor é armazenado nessa localização?

8.4 Para cada uma das seguintes sentenças, escreva uma única instrução que realiza a tarefa especificada. Suponha que as variáveis de ponto flutuante `number1` e `number2` foram declaradas e que `number1` foi inicializado como 7.3. Suponha que a variável `ptr` é do tipo `char *`. Suponha que os arrays `s1` e `s2` são, cada um, arrays `char` de 100 elementos que são inicializados com literais `string`.

- Declare a variável `fPtr` como um ponteiro para um objeto do tipo `double`.
- Atribua o endereço da variável `number1` à variável ponteiro `fPtr`.
- Imprima o valor do objeto apontado por `fPtr`.
- Atribua o valor do objeto apontado por `fPtr` à variável `number2`.
- Imprima o valor de `number2`.
- Imprima o endereço de `number1`.
- Imprima o endereço armazenado em `fPtr`. O valor impresso é o mesmo endereço de `number1`?
- Copie a `string` armazenada no array `s2` para o array `s1`.
- Compare a `string` em `s1` com a `string` em `s2` e imprima o resultado.
- Acréscime os primeiros 10 caracteres da `string` em `s2` à `string` em `s1`.
- Determine o comprimento da `string` em `s1` e imprima o resultado.
- Atribua à `ptr` a localização do primeiro token em `s2`. Os delimitadores de tokens são vírgulas (,).

8.5 Realize a tarefa especificada em cada uma das seguintes instruções:

- Escreva o cabeçalho de função para uma função chamada `exchange` que aceita dois ponteiros para números de dupla precisão com ponto flutuante `x` e `y` como parâmetros e que não retorna um valor.
- Escreva o protótipo de função para a função na parte (a).
- Escreva o cabeçalho de função para uma função chamada `evaluate` que retorna um inteiro e aceita como parâmetros o inteiro `x` e um ponteiro para a função `poly`. A função `poly` aceita um parâmetro do tipo inteiro e retorna um inteiro.
- Escreva o protótipo de função para a função na parte (c).
- Escreva duas instruções que inicializam, cada uma, o array de caractere `vowel` com a `string` de vogais, "AEIOU".

8.6 Encontre o erro em cada um dos seguintes segmentos de programa. Suponha as seguintes declarações e instruções:

```
int *zPtr;           // zPtr irá referenciar o array z
int *aPtr = 0;
void *sPtr = 0;
int number;
int z[5] = { 1, 2, 3, 4, 5 };
```

- `++zPtr;`
- // utiliza ponteiro para obter o primeiro valor de array
`number = zPtr;`
- // atribui o array de 2 elementos (o valor 3) ao número
`number = *zPtr[2];`
- // imprime todo o array `z`
`for ( int i = 0; i <= 5; i++ )`
`cout << zPtr[i] << endl;`
- // atribui o valor apontado por `sPtr` ao número
`number = *sPtr;`

```
f) ++z;
g) char s[ 10 ];
   cout << strncpy( s, "hello", 5 ) << endl;
h) char s[ 12 ];
   strcpy( s, "Welcome Home" );
i) if ( strcmp( string1, string2 ) )
   cout << "The strings are equal" << endl;
```

8.7 O que (se houver algo) é impresso quando cada uma das seguintes instruções é realizada? Se a instrução contiver um erro, descreva o erro e indique como corrigi-la. Suponha as seguintes declarações de variável:

```
char s1[ 50 ] = "jack";
char s2[ 50 ] = "jill";
char s3[ 50 ];
```

```
a) cout << strcpy( s3, s2 ) << endl;
b) cout << strcat( strcat( strcpy( s3, s1 ), " and " ), s2 )
   << endl;
c) cout << strlen( s1 ) + strlen( s2 ) << endl;
d) cout << strlen( s3 ) << endl;
```

Respostas dos exercícios de revisão

8.1 a) endereço. b) 0, NULL, um endereço. c) 0.

8.2 a) Falsa. O operando do operador de endereço deve ser um *lvalue*; o operador de endereço não pode ser aplicado a constantes ou expressões que não resultam em referências.
 b) Falsa. Um ponteiro para void não pode ser desreferenciado. Esse ponteiro não tem um tipo que permite ao compilador determinar o número de bytes de memória a desreferenciar e o tipo dos dados para os quais o ponteiro aponta.
 c) Falsa. Qualquer tipo de ponteiro pode ser atribuído a ponteiros void. Os ponteiros do tipo void podem ser atribuídos a ponteiros de outros tipos somente com uma coerção de tipo explícita.

8.3 a) `double numbers[ SIZE ] = { 0.0, 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9 };`
 b) `double *nPtr;`
 c) `cout << fixed << showpoint << setprecision( 1 );`
 `for ( int i = 0; i < SIZE; i++ )`
 `cout << numbers[ i ] << ' ';`
 d) `nPtr = numbers;`
 `nPtr = &numbers[ 0 ];`
 e) `cout << fixed << showpoint << setprecision( 1 );`
 `for ( int j = 0; j < SIZE; j++ )`
 `cout << *( nPtr + j ) << ' ';`
 f) `cout << fixed << showpoint << setprecision( 1 );`
 `for ( int k = 0; k < SIZE; k++ )`
 `cout << *( numbers + k ) << ' ';`
 g) `cout << fixed << showpoint << setprecision( 1 );`
 `for ( int m = 0; m < SIZE; m++ )`
 `cout << nPtr[ m ] << ' ';`
 h) `numbers[ 3 ]`
 `*( numbers + 3 )`
 `nPtr[ 3 ]`
 `*( nPtr + 3 )`
 i) O endereço é $1002500 + 8 * 8 = 1002564$. O valor é 8.8.
 j) O endereço de `numbers[ 5 ]` é $1002500 + 5 * 8 = 1002540$.
 O endereço de `nPtr - 4` é $1002540 - 4 * 8 = 1002508$.
 O valor nessa localização é 1.1.

8.4 a) `double *fPtr;`
 b) `fPtr = &number1;`
 c) `cout << "The value of *fPtr is " << *fPtr << endl;`
 d) `number2 = *fPtr;`
 e) `cout << "The value of number2 is " << number2 << endl;`
 f) `cout << "The address of number1 is " << &number1 << endl;`

- g) `cout << "The address stored in fPtr is " << fPtr << endl;`
Sim, o valor é o mesmo.
- h) `strcpy( s1, s2 );`
- i) `cout << "strcmp(s1, s2) = " << strcmp( s1, s2 ) << endl;`
- j) `strncat( s1, s2, 10 );`
- k) `cout << "strlen(s1) = " << strlen( s1 ) << endl;`
- l) `ptr = strtok( s2, ", " );`

- 8.5**
- a) `void exchange( double *x, double *y )`
 - b) `void exchange( double *, double * );`
 - c) `int evaluate( int x, int (*poly)( int ) )`
 - d) `int evaluate( int, int (*)( int ) );`
 - e) `char vowel[] = "AEIOU";`
`char vowel[] = { 'A', 'E', 'I', 'O', 'U', '\0' };`

- 8.6**
- a) Erro: `zPtr` não foi inicializado.
Correção: Inicialize `zPtr` com `zPtr = z;`
 - b) Erro: O ponteiro não é desreferenciado.
Correção: Altere a instrução para `number = *zPtr;`
 - c) Erro: `zPtr[ 2 ]` não é um ponteiro e não deve ser desreferenciado.
Correção: altere `*zPtr[ 2 ]` para `zPtr[ 2 ]`.
 - d) Erro: Referir-se a um elemento de array fora dos limites de array com subscrito de ponteiro.
Correção: Para impedir isso, mude o operador relacional na instrução `for` para `<`.
 - e) Erro: Desreferenciar um ponteiro `void`.
Correção: Para desreferenciar o ponteiro `void`, ele deve primeiro sofrer coerção para um ponteiro do tipo inteiro. Mude a instrução para `number = *static_cast< int * >( sPtr );`
 - f) Erro: Tentar modificar um nome de array com a aritmética de ponteiros.
Correção: Utilize uma variável ponteiro em vez do nome de array para realizar a aritmética de ponteiros ou utilize subscritos no nome de array para referenciar um elemento específico.
 - g) Erro: A função `strncpy` não escreve um caractere de terminação nulo para o array `s`, porque seu terceiro argumento é igual ao comprimento da string `"hello"`.
Correção: Torne 6 o terceiro argumento de `strncpy` ou atribua `'\0'` a `s[ 5 ]` para assegurar que o caractere de terminação nulo seja adicionado à string.
 - h) Erro: O array de caractere `s` não é grande o bastante para armazenar o caractere de terminação nulo.
Correção: Declare o array com mais elementos.
 - i) Erro: A função `strcmp` retornará 0 se as strings forem iguais; portanto, a condição na instrução `if` será falsa e a instrução de saída não será executada.
Correção: Compare explicitamente o resultado de `strcmp` com o 0 na condição da instrução `if`.

- 8.7**
- a) `jill`
 - b) `jack and jill`
 - c) 8
 - d) 13

Exercícios

- 8.8** Determine se as seguintes sentenças são *verdadeiras* ou *falsas*. Se *falsa*, explique por quê.
- a) Dois ponteiros que apontam para arrays diferentes não podem ser comparados significativamente.
 - b) Como o nome de um array é um ponteiro para o primeiro elemento do array, os nomes de array podem ser manipulados precisamente da mesma maneira que os ponteiros.
- 8.9** Para cada um dos itens a seguir, escreva instruções C++ que realizam a tarefa especificada. Suponha que inteiros sem sinal estejam armazenados em dois bytes e que o endereço inicial do array esteja na posição 1002500 da memória.
- a) Declare um array do tipo `unsigned int` chamado `values` com cinco elementos e inicialize os elementos para os inteiros pares de 2 a 10. Suponha que a constante simbólica `SIZE` foi definida como 5.
 - b) Declare um ponteiro `vPtr` que aponta para um objeto do tipo `unsigned int`.
 - c) Utilize uma instrução `for` para imprimir os elementos do array `values` usando notação de array subscrito.
 - d) Escreva duas instruções separadas que atribuem o endereço inicial do array `values` à variável ponteiro `vPtr`.
 - e) Utilize uma instrução `for` para imprimir os elementos do array `values` utilizando a notação de ponteiro/deslocamento.
 - f) Utilize uma instrução `for` para imprimir os elementos do array `values` utilizando a notação de ponteiro/deslocamento com o nome de array como o ponteiro.

- g) Utilize uma instrução `for` para imprimir os elementos do array `values` utilizando subscritos no ponteiro para o array.
- h) Referencie o quinto elemento de `values` utilizando a notação de subscrito de array, a notação de ponteiro/deslocamento com o nome de array como o ponteiro, a notação de subscrito de ponteiro e a notação de ponteiro/deslocamento.
- i) Que endereço é referenciado por `vPtr + 3`? Que valor é armazenado nessa localização?
- j) Supondo que `vPtr` aponte para `values[ 4 ]`, que endereço é referenciado por `vPtr -= 4`? Que valor é armazenado nessa localização?

8.10 Para cada um dos itens seguintes, escreva uma única instrução que realiza a tarefa indicada. Suponha que as variáveis do tipo inteiro `long` `value1` e `value2` tenham sido declaradas e que `value1` tenha sido inicializado como 200000.

- a) Declare a variável `longPtr` como um ponteiro para um objeto do tipo `long`.
- b) Atribua o endereço da variável `value1` à variável ponteiro `longPtr`.
- c) Imprima o valor do objeto apontado por `longPtr`.
- d) Atribua o valor do objeto apontado por `longPtr` à variável `value2`.
- e) Imprima o valor de `value2`.
- f) Imprima o endereço de `value1`.
- g) Imprima o endereço armazenado em `longPtr`. O valor impresso é o mesmo que o endereço de `value1`?

8.11 Realize a tarefa especificada por cada uma das seguintes instruções:

- a) Escreva o cabeçalho de função para a função `zero` que aceita um parâmetro de array `bigIntegers` do tipo inteiro `long` e que não retorna um valor.
- b) Escreva o protótipo de função para a função na parte (a).
- c) Escreva o cabeçalho de função para a função `add1AndSum` que aceita um parâmetro de array `oneTooSmall` do tipo inteiro e que retorna um inteiro.
- d) Escreva o protótipo de função para a função descrita na parte (c).

Nota: Os exercícios 8.12 a 8.15 são razoavelmente desafiadores. Uma vez que tiver resolvido esses problemas, você deve ser capaz de implementar muitos jogos de cartas populares.

8.12 Modifique o programa na Figura 8.27 de modo que a função de distribuição de cartas distribua uma mão de pôquer de cinco cartas. Então escreva as funções para realizar cada uma das seguintes tarefas:

- a) Determine se a mão contém um par.
- b) Determine se a mão contém dois pares.
- c) Determine se a mão contém uma trinca (por exemplo, três valetes).
- d) Determine se a mão contém uma quadra (por exemplo, quatro ases).
- e) Determine se a mão contém um *flush* (isto é, todas cinco cartas do mesmo naipe).
- f) Determine se a mão contém um *straight* (isto é, cinco cartas de valores consecutivos).

8.13 Utilize as funções desenvolvidas no Exercício 8.12 para escrever um programa que distribua duas mãos de pôquer de cinco cartas, avalie cada mão e determine qual é a melhor mão.

8.14 Modifique o programa desenvolvido no Exercício 8.13 de modo que você possa simular o carteador. A mão de cinco cartas do carteador é distribuída ‘no escuro’, então o jogador não pode vê-la. O programa deve então avaliar a mão do carteador e, com base na qualidade da mão, o carteador deve distribuir uma, duas ou três mais cartas para substituir o número correspondente de cartas desnecessárias na mão original. O programa então deve reavaliar a mão do carteador. [Atenção: Este é um problema difícil!]

8.15 Modifique o programa desenvolvido no Exercício 8.14 de modo que ele trate a mão do carteador, mas permita ao jogador decidir quais cartas ele quer substituir. O programa então deve avaliar ambas as mãos e determinar quem ganha. Agora utilize esse novo programa para disputar 20 jogos contra o computador. Quem ganha mais jogos, você ou o computador? Faça um de seus amigos disputar 20 jogos contra o computador. Quem ganha mais jogos? Com base nos resultados desses jogos, faça modificações apropriadas para refinar seu programa de jogar de pôquer. [Nota: Este também é um problema difícil.] Dispute mais 20 jogos. Seu programa modificado reproduziu um jogo melhor?

8.16 No programa de embaralhamento e distribuição de cartas das figuras 8.25–8.27, utilizamos intencionalmente um algoritmo ineficiente de embaralhamento que introduziu a possibilidade de adiamento indefinido. Nesse problema, você criará um algoritmo de embaralhamento de alto desempenho que evita o adiamento indefinido.

Modifique as figuras 8.25–8.27 como mostrado a seguir. Inicialize o array `deck` como mostrado na Figura 8.36. Modifique a função `shuffle` para iterar linha por linha e coluna por coluna pelo array, tocando uma vez em cada elemento. Cada elemento deve ser trocado por um elemento do array aleatoriamente selecionado. Imprima o array resultante para determinar se o baralho é embaralhado satisfatoriamente (como na Figura 8.37, por exemplo). Você pode querer que o programa chame a função `shuffle` várias vezes para assegurar um embaralhamento satisfatório.

Observe que, embora a abordagem nesse problema melhore o algoritmo de embaralhamento, o algoritmo de distribuição de cartas ainda requer pesquisar o array `deck` para encontrar a carta 1, depois a carta 2, a carta 3 e assim por diante. Pior ainda, mesmo depois de o

Array deck não embaralhado

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	14	15	16	17	18	19	20	21	22	23	24	25	26
2	27	28	29	30	31	32	33	34	35	36	37	38	39
3	40	41	42	43	44	45	46	47	48	49	50	51	52

Figura 8.36 Array deck não embaralhado.

Exemplo do array deck embaralhado

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	19	40	27	25	36	46	10	34	35	41	18	2	44
1	13	28	14	16	21	30	8	11	31	17	24	7	1
2	12	33	15	42	43	23	45	3	29	32	4	47	26
3	50	38	52	39	48	51	9	5	37	49	22	6	20

Figura 8.37 Exemplo do array deck embaralhado.

algoritmo de distribuição de cartas localizar e distribuir a carta, ele continua pesquisando pelo restante do baralho. Modifique o programa das figuras 8.25–8.27 de modo que, uma vez que uma carta é distribuída, nenhuma tentativa adicional de localizar esse número de carta seja feita, e o programa imediatamente prossiga com a distribuição da próxima carta.

8.17 (*Simulação: A tartaruga e a lebre*) Neste exercício, você recriará a clássica corrida da tartaruga e da lebre. Você utilizará geração de números aleatórios para desenvolver uma simulação desse memorável evento.

Nossos competidores começam a corrida no ‘quadrado 1’ de 70 quadrados. Cada quadrado representa uma possível posição ao longo do percurso da competição. A linha de chegada está no quadrado 70. O primeiro competidor a alcançar ou passar o quadrado 70 é recompensado com um cesto de cenouras frescas e alface. O percurso envolve subir uma montanha escorregadia, então ocasionalmente os competidores perdem terreno.

Há um relógio que toca uma vez por segundo. A cada tique do relógio, seu programa deve ajustar a posição dos animais de acordo com as regras na Figura 8.38.

Animal	Tipo de movimento	Porcentagem do tempo	Movimento real
Tartaruga	Caminhada rápida	50%	3 quadrados à direita
	Escorregão	20%	6 quadrados à esquerda
	Caminhada lenta	30%	1 quadrado à direita
Lebre	Parada	20%	Nenhum movimento
	Salto grande	20%	9 quadrados à direita
	Escorregão grande	10%	12 quadrados à esquerda
	Salto pequeno	30%	1 quadrado à direita
	Escorregão pequeno	20%	2 quadrados à esquerda

Figura 8.38 Regras para mover a tartaruga e a lebre.

Utilize variáveis para monitorar a posição dos animais (isto é, os números de posição são 1–70). Inicie cada animal na posição 1 (isto é, a ‘partida’). Se um animal escorrega para a esquerda antes do quadrado 1, mova o animal de volta para quadrado 1.

Gere as porcentagens da tabela precedente produzindo um inteiro aleatório i no intervalo $1 \leq i \leq 10$. Para a tartaruga, realize uma ‘caminhada rápida’ quando $1 \leq i \leq 5$, um ‘escorregão’ quando $6 \leq i \leq 7$ ou uma ‘caminhada lenta’ quando $8 \leq i \leq 10$. Utilize uma técnica semelhante para mover a lebre.

Inicie a corrida imprimindo

```
BANG !!!!!
AND THEY'RE OFF !!!!!
[BANG !!!!!
E LÁ SE VÃO ELES !!!!!]
```

Para cada tique do relógio (isto é, cada repetição de um loop), imprima uma linha de 70 posições para mostrar a letra T na posição da tartaruga e a letra H na posição da lebre. Ocasionalmente, os competidores cairão no mesmo quadrado. Nesse caso, a tartaruga morde a lebre e seu programa deve imprimir OUCH!!! (Ai!) começando nessa posição. Todas as outras posições diferentes de T, H e o OUCH!!! (no caso de um empate) devem estar em branco.

Depois de imprimir cada linha, teste se algum animal alcançou ou passou o quadrado 70. Em caso positivo, imprima o vencedor e termine a simulação. Se a tartaruga ganhar, imprima TORTOISE WINS!!! YAY!!! [A TARTARUGA GANHOU!!! ÊHH!!!]. Se a lebre ganhar, imprima Hare wins. Yuch. [A Lebre ganhou. Uhh.]. Se ambos tiverem a mesma marcação no relógio, você pode querer favorecer a tartaruga (a ‘coitadinha’) ou querer imprimir It's a tie [Empate]. Se nenhum dos animais ganhar, realize o loop novamente para simular o próximo tique do relógio. Quando você estiver pronto para executar seu programa, monte um grupo de fãs para observar a corrida. Você ficará surpreso com o envolvimento da sua audiência!

Seção especial: construindo seu próprio computador

Nos próximos problemas, nós nos desviamos temporariamente do mundo da linguagem de programação de alto nível. Vamos ‘abrir’ um computador e examinar sua estrutura interna. Introduzimos programação de linguagem de máquina e escrevemos vários programas de linguagem de máquina. Para tornar essa uma experiência especialmente valiosa, construímos em seguida um computador (utilizando a *simulação* baseada em software) em que você pode executar seus programas de linguagem de máquina!

8.18 (*Programação de linguagem de máquina*) Vamos criar um computador que chamaremos de Simpletron. Como seu nome implica, é uma máquina simples, mas como logo veremos também é uma máquina poderosa. O Simpletron executa programas escritos na única linguagem que ele entende diretamente, isto é, a Simpletron Machine Language ou, abreviadamente, SML.

O Simpletron contém um *acumulador* — um ‘registrador especial’ em que as informações são colocadas antes de o Simpletron utilizá-las em cálculos ou examiná-las de várias maneiras. Todas as informações no Simpletron são tratadas em termos de *palavras*. Uma palavra é um número decimal de quatro dígitos com sinal como +3364, -1293, +0007, -0001 etc. O Simpletron é equipado com uma memória de 100 palavras e essas palavras são referenciadas por seus números de posição 00, 01, ..., 99.

Antes de executar um programa de SML, devemos *carregar*, ou colocar, o programa na memória. A primeira instrução de cada programa de SML sempre é colocada na posição 00. O simulador começará a executar nessa posição.

Cada instrução escrita em SML ocupa uma palavra da memória do Simpletron; portanto, as instruções são números decimais de quatro dígitos com sinal. Suponha que o sinal de uma instrução de SML seja sempre positivo, mas o sinal de uma palavra de dados pode ser positivo ou negativo. Cada localização na memória de Simpletron pode conter uma instrução, um valor de dados utilizado por um programa ou uma área da memória não-utilizada (e portanto indefinida). Os primeiros dois dígitos de cada instrução do SML são o *código de operação* que especifica a operação a ser realizada. Os códigos de operação SML são mostrados na Figura 8.39.

Os últimos dois dígitos de uma instrução de SML são *operandos* — o endereço da posição da memória contendo a palavra à qual a operação se aplica.

Agora vamos considerar dois programas SML simples. O primeiro programa SML (Figura 8.40) lê dois números do teclado e calcula e imprime sua soma. A instrução +1007 lê o primeiro número do teclado e o coloca na posição 07 (que foi inicializada como zero). A instrução +1008 lê o próximo número na posição 08. A instrução *load*, +2007, coloca (copia) o primeiro número no acumulador; e a instrução *add*, +3008, adiciona o segundo número ao número no acumulador. *Todas as instruções aritméticas da SML deixam seus resultados no acumulador.* A instrução *store*, +2109, coloca (copia) o resultado de volta na posição da memória 09. Então a instrução *write*, +1109, pega o número e o imprime (como um número decimal de quatro dígitos com sinal). A instrução *halt*, +4300, termina a execução.

O programa SML na Figura 8.41 lê dois números a partir do teclado, então determina e imprime o maior valor. Note o uso da instrução +4107 como uma transferência condicional de controle, muito parecida com a instrução *if* do C++.

Código de operação	Significado
<i>Operações de entrada/saída</i>	
<code>const int READ = 10;</code>	Lê uma palavra do teclado para uma posição específica da memória.
<code>const int WRITE = 11;</code>	Escreve na tela uma palavra de uma posição específica da memória.
<i>Operações de carregamento e armazenamento</i>	
<code>const int LOAD = 20;</code>	Carrega uma palavra de uma posição específica na memória para o acumulador.
<code>const int STORE = 21;</code>	Armazena uma palavra do acumulador para uma posição específica na memória.
<i>Operações aritméticas</i>	
<code>const int ADD = 30;</code>	Adiciona uma palavra de uma posição específica na memória à palavra no acumulador (deixa o resultado no acumulador).
<code>const int SUBTRACT = 31;</code>	Subtrai uma palavra de uma posição específica na memória da palavra no acumulador (deixa o resultado no acumulador).
<code>const int DIVIDE = 32;</code>	Divide uma palavra de uma posição específica na memória pela palavra no acumulador (deixa o resultado no acumulador).
<code>const int MULTIPLY = 33;</code>	Multiplica uma palavra de uma posição específica na memória pela palavra no acumulador (deixa o resultado no acumulador).
<i>Operações de transferência de controle</i>	
<code>const int BRANCH = 40;</code>	Desvia para uma posição específica na memória.
<code>const int BRANCHNEG = 41;</code>	Desvia para uma posição específica na memória se o acumulador for negativo.
<code>const int BRANCHZERO = 42;</code>	Desvia para uma posição específica na memória se o acumulador for zero.
<code>const int HALT = 43;</code>	Suspende — o programa completou sua tarefa.

Figura 8.39 Códigos de operação de Simpletron Machine Language (SML).

Posição	Número	Instrução
00	+1007	(Read A)
01	+1008	(Read B)
02	+2007	(Load A)
03	+3008	(Add B)
04	+2109	(Store C)
05	+1109	(Write C)
06	+4300	(Halt)
07	+0000	(Variable A)
08	+0000	(Variable B)
09	+0000	(Result C)

Figura 8.40 Exemplo de SML I.

Posição	Número	Instrução
00	+1009	(Read A)
01	+1010	(Read B)
02	+2009	(Load A)
03	+3110	(Subtract B)
04	+4107	(Desvio negativo para 07)
05	+1109	(Write A)
06	+4300	(Halt)
07	+1110	(Write B)
08	+4300	(Halt)
09	+0000	(Variable A)
10	+0000	(Variable B)

Figura 8.41 Exemplo de SML 2.

Agora escreva programas de SML para realizar cada uma das seguintes tarefas:

- Utilize um loop controlado por sentinela para ler números positivos e calcular e imprimir sua soma. Termine a entrada quando um número negativo for inserido.
- Utilize um loop controlado por contador para ler sete números, alguns negativos e alguns positivos e compute e imprima sua média.
- Leia uma série de números, e determine e imprima o maior número. O primeiro número lido indica quantos números devem ser processados.

8.19 (*Simulador de computador*) À primeira vista, pode parecer pretensioso, mas nesse problema você construirá seu próprio computador. Não, você não irá soldar componentes. Em vez disso, você utilizará a poderosa técnica de *simulação baseada em software* para criar um *modelo de software* do Simpletron. Você não se decepcionará. Seu Simpletron Simulator transformará o computador que você está utilizando em um Simpletron, e você realmente será capaz de executar, testar e depurar os programas de SML escritos no Exercício 8.18.

Quando você executar seu simulador Simpletron, ele deve começar imprimindo

```

*** Bem vindo ao Simpletron!           ***
*** Por favor insira uma instrução      ***
*** (ou data word) por vez em seu programa. Eu vou digitar ***
*** o número de alocação e o ponto de interrogação (?).    ***
*** Então você digita a palavra para a alocação.           ***
*** Digite o número -99999 para parar de inserir dados     ***
*** no seu programa.                                       ***

```

Seu programa deve simular a memória do Simpletron com um de um único subscrito, o array memory de 100 elementos. Agora assuma que o simulador está executando e vamos examinar o diálogo à medida que inserimos o programa do Exemplo 2 do Exercício 8.18:

```

00 ? +1009
01 ? +1010
02 ? +2009
03 ? +3110
04 ? +4107
05 ? +1109
06 ? +4300
07 ? +1110
08 ? +4300
09 ? +0000
10 ? +0000
11 ? -99999
*** Program loading completed ***
*** Program execution begins ***

```

Observe que os números à direita de cada ? no diálogo anterior representam as instruções do programa SML inseridos pelo usuário.

O programa de SML agora foi colocado (ou carregado) no array `memory`. Agora o Simpletron executa seu programa SML. A execução inicia com a instrução na posição 00 e, como o C++, continua sequencialmente, a menos que dirigido para alguma outra parte do programa por uma transferência de controle.

Utilize a variável `accumulator` para representar o registrador acumulador. Utilize a variável `counter` para monitorar a posição na memória que contém a instrução sendo realizada. Utilize a variável `operationCode` para indicar a operação que está sendo atualmente realizada (isto é, os dois dígitos esquerdos da palavra da instrução). Utilize a variável `operand` para indicar a posição da memória em que a instrução atual opera. Portanto, `operand` são os dois dígitos mais à direita da instrução sendo atualmente realizada. Não execute instruções diretamente de memória. Mais precisamente, transfira a próxima instrução que será realizada da memória para uma variável chamada `instructionRegister`. Então ‘pegue’ os dois dígitos esquerdos e os coloque em `operationCode` e ‘pegue’ os dois dígitos direitos e os coloque no `operand`. Quando o Simpletron começa a executar, todos os registradores especiais são inicializados como zero.

Agora vamos ‘percorrer’ a execução da primeira instrução de SML, +1009 na posição da memória 00. Isso é chamado um *ciclo de execução de instrução*.

O `counter` informa a posição da próxima instrução que será realizada. Realizamos uma *busca (fetch)* do conteúdo dessa posição a partir de `memory` utilizando a instrução C++

```
instructionRegister = memory[ counter ];
```

O código de operação e o operando são extraídos do registrador de instrução pelas instruções

```
operationCode = instructionRegister / 100;
operand = instructionRegister % 100;
```

Agora o Simpletron deve determinar que o código de operação é na realidade uma *read* (versus um *write*, um *load* etc.) Um `switch` diferencia entre as 12 operações de SML.

Na instrução `switch`, o comportamento de várias instruções SML é simulado como mostrado na Figura 8.42 (deixamos os outros para o leitor).

A instrução *halt* também faz com que o Simpletron imprima o nome e o conteúdo de cada registrador, bem como o conteúdo completo da memória. Esse tipo de impressão é freqüentemente chamado de *dump de computador*. Para ajudá-lo a programar sua função de *dump*, um formato de *dump* de exemplo é mostrado na Figura 8.43. Observe que um *dump*, depois de executar um programa Simpletron, mostraria os valores reais das instruções e os valores dos dados no momento em que a execução terminasse. Para formatar os números com seu sinal como mostrado no *dump*, utilize o manipulador de fluxo `showpos`. Para desativar a exibição do sinal, utilize o manipulador de fluxo `noshwpos`. Para números que tenham menos de quatro dígitos, você pode formatar números com zeros à esquerda entre o sinal e o valor utilizando a seguinte instrução antes de gerar a saída do valor:

```
cout << setfill( '0' ) << internal;
```

O manipulador de fluxo parametrizado `setfill` (do cabeçalho `<iomanip>`) especifica o caractere de preenchimento que irá aparecer entre o sinal e o valor quando um número for exibido com uma largura de campo de cinco caracteres mas sem quatro dígitos. (Uma posição na largura de campo é reservada para o sinal.) O manipulador de fluxo `internal` indica que os caracteres de preenchimento aparecem entre o sinal e o valor numérico.

Vamos prosseguir com a execução da primeira instrução de nosso programa — +1009 na posição 00. Como indicamos, a instrução `switch` simula isso executando a instrução C++.

```
cin >> memory[ operand ];
```

Um ponto de interrogação (?) deve ser exibido na tela antes de a instrução `cin` executar para solicitar a entrada ao usuário. O Simpletron espera o usuário digitar um valor e pressionar a tecla *Enter*. O valor então é lido na posição 09.

<i>read:</i>	<code>cin >> memory[operand];</code>
<i>load:</i>	<code>accumulator = memory[operand];</code>
<i>add:</i>	<code>accumulator += memory[operand];</code>
<i>branch:</i>	Discutiremos as instruções de desvio brevemente.
<i>halt:</i>	Essa instrução imprime a mensagem *** Simpletron execution terminated ***

Figura 8.42 Comportamento de instruções SML.


```

REGISTERS:
accumulator      +0000
counter          00
instructionRegister +0000
operationCode     00
operand          00

MEMORY:
    0  +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
  10  +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
  20  +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
  30  +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
  40  +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
  50  +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
  60  +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
  70  +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
  80  +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000
  90  +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000 +0000

```

Figura 8.43 Um dump de exemplo.

Neste ponto, a simulação da primeira instrução é concluída. Tudo o que resta é preparar o Simpletron para executar a próxima instrução. A instrução que acabamos de realizar não era uma transferência de controle, portanto precisamos meramente incrementar os registradores de contadores de instruções como mostrado a seguir:

```
++counter;
```

Isso completa a execução simulada da primeira instrução. O processo inteiro (isto é, o ciclo de execução de instrução) começa de novo com a busca da próxima instrução a executar.

Agora vamos considerar como simular as instruções de desvio (isto é, as transferências de controle). Tudo o que precisamos fazer é ajustar o valor no contador de instrução apropriadamente. Portanto, a instrução de desvio incondicional (40) é simulada no `switch` como

```
counter = operand;
```

A instrução condicional ‘desvie se acumulador for zero’ é simulada como

```
if ( accumulator == 0 )
    counter = operand;
```

Nesse ponto, você deve implementar seu Simpletron Simulator e executar cada um dos programas em SML que você escreveu no Exercício 8.18. Você pode sofisticar a SML com recursos adicionais e adaptá-los ao seu simulador.

Seu simulador deve verificar vários tipos de erros. Durante a fase de carregamento do programa, por exemplo, cada número que usuário digita no Simpletron memory deve estar no intervalo -9999 a +9999. Seu simulador deve utilizar um `while` para testar se cada número inserido está nesse intervalo e, se não tiver, continuar pedindo para ao usuário tentar novamente o número até inserir um número correto.

Durante a fase de execução, seu simulador deve verificar vários erros sérios, como tentativas de divisão por zero, tentativas de execução de códigos de operação inválidos, estouros de acumulador (isto é, operações aritméticas resultando em valores maiores que +9999 ou menores que -9999) e assim por diante. Os erros sérios são chamados **erros fatais**. Quando um erro fatal é detectado, seu simulador deve imprimir uma mensagem de erro como

```
*** Tentou dividir por zero ***
*** A execução do Simpletron foi interrompida ***
```

e deve imprimir um dump de computador completo no formato discutido previamente. Isso ajudará o usuário localizar o erro no programa.

Mais exercícios sobre ponteiros

8.20 Modifique o programa de embaralhamento e distribuição de cartas das figuras 8.25–8.27 para que as operações de embaralhamento e distribuição sejam realizadas pela mesma função (`shuffleAndDeal`). A função deve conter uma instrução de loop aninhada que seja semelhante à função `shuffle` na Figura 8.26.

8.21 O que esse programa faz?

```

1  // Ex. 8.21: ex08_21.cpp
2  // O que esse programa faz?
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  void mystery1( char *, const char * ); // protótipo
9
10 int main()
11 {
12     char string1[ 80 ];
13     char string2[ 80 ];
14
15     cout << "Enter two strings: ";
16     cin >> string1 >> string2;
17     mystery1( string1, string2 );
18     cout << string1 << endl;
19     return 0; // indica terminação bem-sucedida
20 } // fim de main
21
22 // O que essa função faz?
23 void mystery1( char *s1, const char *s2 )
24 {
25     while ( *s1 != '\0' )
26         ++s1;
27
28     for ( ; *s1 = *s2; s1++, s2++ )
29         ; // estrutura vazia
30 } // fim da função mystery1

```

8.22 O que esse programa faz?

```

1  // Ex. 8.22: ex08_22.cpp
2  // O que esse programa faz?
3  #include <iostream>
4  using std::cout;
5  using std::cin;
6  using std::endl;
7
8  int mystery2( const char * ); // protótipo
9
10 int main()
11 {
12     char string1[ 80 ];
13
14     cout << "Enter a string: ";
15     cin >> string1;
16     cout << mystery2( string1 ) << endl;
17     return 0; // indica terminação bem-sucedida
18 } // fim de main
19
20 // O que essa função faz?
21 int mystery2( const char *s )
22 {

```

```

23     int x;
24
25     for ( x = 0; *s != '\0'; s++ )
26         ++x;
27
28     return x;
29 } // fim da função mystery2

```

8.23 Localize o erro em cada um dos seguintes segmentos de código. Se for possível corrigir o erro, explique como.

- `int *number;`
`cout << number << endl;`
- `double *realPtr;`
`long *integerPtr;`
`integerPtr = realPtr;`
- `int * x, y;`
`x = y;`
- `char s[] = "this is a character array";`
`for ( ; *s != '\0'; s++)`
`cout << *s << ' ';`
- `short *numPtr, result;`
`void *genericPtr = numPtr;`
`result = *genericPtr + 7;`
- `double x = 19.34;`
`double xPtr = &x;`
`cout << xPtr << endl;`
- `char *s;`
`cout << s << endl;`

8.24 (*Quicksort – Técnica de classificação recursiva*) Você viu previamente as técnicas de classificação do tipo *bucket sort* e classificação por seleção. Agora apresentamos a técnica de classificação recursiva chamada Quicksort. O algoritmo básico para um array de um único subscrito de valores é como segue:

- Passo de particionamento:* Pegue o primeiro elemento do array não-classificado e determine sua localização final no array classificado (isto é, todos os valores à esquerda do elemento no array são menores que o elemento e todos os valores à direita do elemento no array são maiores que o elemento). Agora temos um elemento em sua posição adequada e dois subarrays não-classificados.
- Passo recursivo:* Realize o *Passo 1* em cada subarray não-classificado.

Toda vez que o *Passo 1* for realizado em um subarray, outro elemento é colocado em sua posição final no array classificado, e dois subarrays não-classificados são criados. Quando um subarray consiste em um elemento, esse subarray deve ser classificado; portanto, esse elemento está em sua localização final.

O algoritmo básico parece suficientemente simples, mas como determinamos a posição final do primeiro elemento de cada subarray? Como um exemplo, considere o seguinte conjunto de valores (o elemento em negrito é o elemento de particionamento — ele será colocado em sua localização final no array classificado) :

37 2 6 4 89 8 10 12 68 45

- Iniciando do elemento mais à direita do array, compare cada elemento com **37** até um elemento menor que **37** ser encontrado. Então troque **37** e esse elemento. O primeiro elemento menor que **37** é 12, então **37** e 12 são permutados. Os valores agora residem no array como mostrado a seguir:

12 2 6 4 89 8 10 **37** 68 45

O elemento 12 está em itálico para indicar que acabou ser permutado com 37.

- Iniciando à esquerda do array, mas começando com o elemento depois de 12, compare cada elemento com **37** até que um elemento maior que **37** seja localizado. Então troque **37** e esse elemento. O primeiro elemento maior que **37** é 89, então **37** e 89 foram permutados. Os valores agora residem no array como mostrado a seguir:

12 2 6 4 **37** 8 10 89 68 45

- Iniciando à direita, mas começando com o elemento antes de 89, compare cada elemento com **37** até que um elemento menor que **37** seja localizado. Então troque **37** e esse elemento. O primeiro elemento menor que **37** é 10, então **37** e 10 são permutados. Os valores agora residem no array como mostrado a seguir:

12 2 6 4 10 8 **37** 89 68 45

d) Iniciando à esquerda, mas começando com o elemento depois de 10, compare cada elemento com **37** até que elemento maior que **37** seja localizado. Então troque **37** e esse elemento. Não há mais elementos maiores que **37**, então, quando comparamos **37** com ele mesmo sabemos que **37** foi colocado na sua localização final do array classificado.

Depois que o particionamento foi aplicado ao array, restam dois subarrays não classificados. O subarray com valores menores que 37 contém 12, 2, 6, 4, 10 e 8. O subarray com valores maiores que 37 contém 89, 68 e 45. A classificação continua com ambos os subarrays sendo particionados da mesma maneira que o array original.

Com base na discussão precedente, escreva uma função recursiva `quickSort` para classificar um array de inteiros de um único subscrito. A função deve receber como argumentos um array de inteiros, um subscrito inicial e um subscrito final. A função `partition` deve ser chamada por `quickSort` para realizar o passo de particionamento.

8.25 (*Percorrendo um labirinto*) A grade de cerquilhas (#) e pontos (.) na Figura 8.44 é uma representação de um array bidimensional de um labirinto. No array bidimensional, as cerquilhas representam as paredes do labirinto e os pontos representam quadrados nos possíveis caminhos pelo labirinto. Movimentos são permitidos apenas nas posições do array que contiverem um ponto.

Há um algoritmo simples para percorrer um labirinto que garante a localização da saída (supondo que existe uma saída). Se não houver uma saída, você chegará à localização inicial novamente. Coloque a sua mão direita na parede à sua direita e comece a andar para a frente. Nunca tire a sua mão da parede. Se o labirinto virar para a direita, siga a parede à direita. Contanto que você não remova a sua mão da parede, você acabará chegando à saída do labirinto. É possível que haja um caminho mais curto do que o que você tomou, mas a saída do labirinto é garantida se o algoritmo for seguido.

Escreva a função recursiva `mazeTraverse` para percorrer o labirinto. A função deve receber argumentos que incluem um array de 12 por 12 caracteres que representa o labirinto e a localização inicial do labirinto. À medida que `mazeTraverse` tenta localizar a saída do labirinto, ele deve colocar o caractere X em cada quadrado no caminho. A função deve exibir o labirinto depois de cada movimento de modo que o usuário possa observar enquanto o problema da saída do labirinto é resolvido.

8.26 (*Gerando labirintos aleatoriamente*) Escreva uma função `mazeGenerator` que recebe como um argumento um array de 12 caracteres bidimensional e produza aleatoriamente um labirinto. A função também deve fornecer as posições inicial e final do labirinto. Experimente a função `mazeTraverse` do Exercício 8.25, utilizando vários labirintos gerados aleatoriamente.

8.27 (*Labirintos de qualquer tamanho*) Generalize as funções `mazeTraverse` e `mazeGenerator` dos exercícios 8.25 e 8.26 para processar labirintos de qualquer largura e altura.

8.28 (*Modificações para o Simpletron Simulator*) No Exercício 8.19, você escreveu uma simulação de software de um computador que executa programas escritos em Simpletron Machine Language (SML). Nesse exercício, são propostas várias modificações e aprimoramentos para o Simpletron Simulator. Nos exercícios 21.26 e 21.27, propomos construir um compilador que converte programas escritos em uma linguagem de programação de alto nível (uma variação do BASIC) em SML. Algumas das seguintes modificações e melhorias podem ser necessárias para executar os programas produzidos pelo compilador. [Nota: Algumas modificações podem entrar em conflito com outras e, portanto, devem ser feitas separadamente.]

- Estender a memória do Simpletron Simulator para conter 1.000 posições de memória a fim de permitir que o Simpletron trate programas maiores.
- Permitir que o simulador realize cálculos de módulo. Isso requer uma instrução SML adicional.
- Permitir que o simulador realize cálculos de exponenciação. Isso requer uma instrução SML adicional.
- Modificar o simulador para utilizar valores hexadecimais em vez de valores inteiros para representar instruções SML.
- Modificar o simulador para permitir saída de uma nova linha. Isso requer uma instrução SML adicional.

```

# # # # # # # # # #
# . . . # . . . . . #
. . # . # . # # # . #
# # # . # . . . # . #
# . . . . # # # . # .
# # # # . # . # . #
# . . # . # . # . #
# # . # . # . # . #
# . . . . . . # . #
# # # # # . # # # . #
# . . . . . # . . . #
# # # # # # # # # #

```

Figura 8.44 Representação de um labirinto com um array bidimensional.

- f) Modificar o simulador para processar valores de ponto flutuante além de valores inteiros.
- g) Modificar o simulador para tratar entrada de string. [Dica: Cada palavra do Simpletron pode ser dividida em dois grupos, cada um armazenando um inteiro de dois dígitos. Cada inteiro de dois dígitos representa o equivalente ASCII decimal de um caractere. Adicione uma instrução de linguagem de máquina que irá inserir uma string e armazenar a string inicial em uma posição da memória específica do Simpletron. A primeira metade da palavra nessa posição será uma contagem do número de caracteres na string (isto é, o comprimento da string). Cada meia-palavra sucessiva contém um caractere ASCII como dois dígitos decimais expressos. A instrução de linguagem de máquina converte cada caractere em seu equivalente ASCII e atribui a ele uma meia-palavra.]
- h) Modificar o simulador para tratar saída de strings armazenadas no formato da parte (g). [Dica: Adicione uma instrução de linguagem de máquina que imprimirá uma string inicial em certa posição da memória de Simpletron. A primeira metade da palavra nessa posição é uma contagem do número de caracteres na string (isto é, o comprimento da string). Cada meia-palavra sucessiva contém um caractere ASCII como dois dígitos decimais expressos. Cada meia-palavra sucessiva contém um caractere de ASCII como dois dígitos decimais expressos.]
- i) Modifique o simulador para incluir a instrução SML_DEBUG que imprime um dump de memória depois que cada instrução executa. Forneça ao SML_DEBUG um código de operação de 44. A palavra +4401 ativa o modo de depuração, e +4400 o desativa.

8.29 O que esse programa faz?

```

1 // Ex. 8.29: ex08_29.cpp
2 // O que esse programa faz?
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 bool mystery3( const char *, const char * ); // protótipo
9
10 int main()
11 {
12     char string1[ 80 ], string2[ 80 ];
13
14     cout << "Enter two strings: ";
15     cin >> string1 >> string2;
16     cout << "The result is " << mystery3( string1, string2 ) << endl;
17     return 0; // indica terminação bem-sucedida
18 } // fim de main
19
20 // O que essa função faz?
21 bool mystery3( const char *s1, const char *s2 )
22 {
23     for ( ; *s1 != '\0' && *s2 != '\0'; s1++, s2++ )
24
25         if ( *s1 != *s2 )
26             return false;
27
28     return true;
29 } // fim da função mystery3

```

Exercícios de manipulação de string

[Nota: Os exercícios a seguir devem ser implementados utilizando strings baseadas em ponteiro no estilo C.]

- 8.30** Escreva um programa que utiliza a função `strcmp` para comparar duas strings inseridas pelo usuário. O programa deve declarar se a primeira string é menor que, igual a ou maior que a segunda string.
- 8.31** Escreva um programa que utiliza a função `strncmp` para comparar duas strings inseridas pelo usuário. O programa deve inserir o número de caracteres a ser comparado. O programa deve declarar se a primeira string é menor que, igual a ou maior que a segunda string.
- 8.32** Escreva um programa que utiliza a geração de números aleatórios para criar frases. O programa deve utilizar quatro arrays de ponteiros para `char` chamados `article`, `noun`, `verb` e `preposition`. O programa deve criar uma frase selecionando aleatoriamente uma palavra de cada array na seguinte ordem: `article`, `noun`, `verb`, `preposition`, `article` e `noun`. Quando cada palavra é selecionada, ela deve

ser concatenada pelas palavras anteriores em um array suficientemente grande para armazenar a frase inteira. As palavras devem ser separadas por espaços. Quando a frase final for enviada para saída, ela deve iniciar com uma letra maiúscula e terminar com um ponto. O programa deve gerar 20 dessas frases.

Os arrays devem ser preenchidos como segue: O array `article` deve conter os artigos "the", "a", "one", "some" e "any"; e array `noun` deve conter os nomes "boy", "girl", "dog", "town" e "car"; o array `verb` deve conter os verbos "drove", "jumped", "ran", "walked" e "skipped"; o array `preposition` deve conter as preposições "to", "from", "over", "under" e "on".

Depois de concluir o programa, modifique-o para produzir uma pequena história com várias dessas frases. (E que tal um escritor de teses aleatório?!)

- 8.33** (*Limericks*) Um limerick é um poema humorístico de cinco versos em que a primeira e a segunda linha rimam com a quinta, e a terceira linha rima com a quarta. Utilizando técnicas semelhantes às aquelas desenvolvidas no Exercício 8.32, escreva um programa C++ que produz limericks aleatórios. Aprimorar esse programa para produzir bons limericks é um problema desafiador, mas o resultado vale o esforço!
- 8.34** Escreva um programa que codifica frases da língua inglesa em 'latim de porco'. O latim de porco é uma forma de linguagem codificada freqüentemente utilizada por diversão. Existem muitas variações nos métodos utilizados para formar frases em latim de porco. Para simplificar, utilize o seguinte algoritmo: Para formar uma frase em latim de porco a partir do inglês, tokenize a frase em palavras com função `strtok`. Para traduzir cada palavra inglesa em uma palavra do latim de porco, coloque a primeira letra da palavra inglesa no final da palavra e adicione as letras "ay". Portanto, a palavra "jump" torna-se "umpjay", a palavra "the" torna-se "hetay" e a palavra "computer" torna-se "omputercay". Os espaços entre as palavras permanecem iguais. Suponha que a escrita do inglês consista em palavras separadas por espaços, não haja nenhuma marcação de pontuação e todas as palavras tenham duas ou mais letras. A função `printLatinWord` deve exibir cada palavra. [*Dica:* Toda vez que um token for localizado em uma chamada para `strtok`, passe o ponteiro do token para a função `printLatinWord` e imprima a palavra em latim de porco.]
- 8.35** Escreva um programa que insere um número de telefone como uma string na forma (555) 555-5555. O programa deve utilizar a função `strtok` para extrair o código de área como um token, os três primeiros dígitos do número de telefone como um segundo token e os últimos quatro dígitos do número de telefone como um terceiro token. Os sete dígitos do número de telefone devem ser concatenados em uma string. O código de área e o número de telefone devem ser impressos.
- 8.36** Escreva um programa que insere uma linha de texto, tokeniza a linha com a função `strtok` e gera saída dos tokens na ordem inversa.
- 8.37** Utilize as funções de comparação de string discutidas na Seção 8.13.2 e as técnicas para classificar arrays desenvolvidas no Capítulo 7 para escrever um programa que alfabeta uma lista de strings. Utilize os nomes de 10 ou 15 bairros em sua área como dados para seu programa.
- 8.38** Escreva duas versões de cada função de cópia de string e concatenação de string na Figura 8.30. A primeira versão deve utilizar o subscrito de array, e a segunda, ponteiros e aritmética de ponteiros.
- 8.39** Escreva duas versões de cada função de comparação de string na Figura 8.30. A primeira versão deve utilizar o subscrito de array, e a segunda, ponteiros e aritmética de ponteiros.
- 8.40** Escreva duas versões da função `strlen` na Figura 8.30. A primeira versão deve utilizar o subscrito de array, e a segunda, ponteiros e aritmética de ponteiros.

Seção especial: exercícios avançados de manipulação de string

Os exercícios precedentes são voltados para texto e projetados para testar o entendimento do leitor de conceitos fundamentais de manipulação de string. Esta seção inclui uma coleção de exercícios de manipulação de string avançados e intermediários. O leitor deve achar esses problemas desafiadores, e até divertidos. Os problemas variam consideravelmente em dificuldade. Alguns requerem uma hora ou duas para escrever e implementar o programa. Outros são úteis para atribuições de laboratório que talvez requeiram duas ou três semanas de estudo e implementação. Alguns são projetos de conclusão de curso desafiadores.

- 8.41** (*Análise de texto*) A disponibilidade de computadores com capacidades de manipulação de string resultou em algumas abordagens bastante interessantes para analisar textos de grandes autores. Muita atenção foi dada à polêmica de que William Shakespeare não teria existido de fato. Alguns especialistas acreditam que há evidência substancial para indicar que Christopher Marlowe ou outros foram os verdadeiros autores das obras-primas atribuídas a Shakespeare. Os pesquisadores têm utilizado computadores para encontrar semelhanças na escrita desses autores. Esse exercício examina três métodos para analisar textos com um computador. Observe que milhares de textos, inclusive os de Shakespeare, estão disponíveis on-line em www.gutenberg.org.

- a) Escreva um programa que lê várias linhas de texto do teclado e imprime uma tabela que indica o número de ocorrências de cada letra do alfabeto no texto. Por exemplo, a frase

To be, or not to be: that is the question:

contém um 'a', dois 'b', nenhum 'c' etc.

- b) Escreva um programa que lê várias linhas de texto e imprime uma tabela que indica o número de palavras de uma letra, palavras de duas letras, palavras de três letras etc. que aparecem no texto. Por exemplo, a frase

Whether 'tis nobler in the mind to suffer
 contém os seguintes comprimentos de palavra e ocorrências:

Comprimento de palavra	Ocorrências
1	0
2	2
3	1
4	2 (incluindo 'tis)
5	0
6	2
7	1

- c) Escreva um programa que lê várias linhas de texto e imprime uma tabela que indica o número de ocorrências de cada palavra diferente no texto. A primeira versão de seu programa deve incluir as palavras na tabela na mesma ordem em que elas aparecem no texto. Por exemplo, as linhas

To be, or not to be: that is the question:
 Whether 'tis nobler in the mind to suffer

Contém a palavra 'to' três vezes, a palavra 'be' duas vezes, a palavra 'or' uma vez etc. Então você deve tentar uma impressão mais interessante (e útil) em que as palavras são classificadas alfabeticamente.

- 8.42** (*Processamento de textos*) Uma função importante em sistemas de processamento de texto é a *justificação de texto* — que é o alinhamento à esquerda e à direita de uma página simultaneamente. Isso gera um documento com um *layout* profissional que dá a impressão de que o documento foi editorado em vez de preparado em uma máquina de escrever. A justificação de texto pode ser realizada em sistemas de computador inserindo caracteres de espaço em branco entre cada uma das palavras em uma linha para que a última palavra de cada linha seja alinhada com a margem direita.

Escreva um programa que lê várias linhas de texto e imprime esse texto no formato alinhado. Suponha que o texto deva ser impresso no papel de 8 1/2 polegadas de largura e que as margens de uma polegada sejam adotadas nos lados esquerdo e direito. Suponha que o computador imprima 10 caracteres por polegada horizontal. Portanto, seu programa deve imprimir 6-1/2 polegadas de texto ou 65 caracteres por linha.

- 8.43** (*Imprimindo datas em vários formatos*) As datas são comumente impressas em vários formatos diferentes na correspondência comercial. Dois dos formatos mais comuns em inglês são

07/21/1955
 July 21, 1955

Escreva um programa que lê uma data no primeiro formato e imprime essa data no segundo formato.

- 8.44** (*Proteção de cheque*) Os computadores são freqüentemente empregados em sistemas de verificação de cheque como aplicativos de folha de pagamento e contas a pagar. Circulam muitas histórias estranhas relacionadas com cheques de pagamento de salário semanal impressos (por equívoco) com quantias acima de \$ 1 milhão. Valores de cheque absurdos são impressos por sistemas computadorizados de preenchimento de cheque devido a erro humano ou falha mecânica. Os projetistas de sistemas embutem controles em seus sistemas para evitar a emissão desses cheques errados.

Outro problema sério é a alteração intencional de um valor do cheque por alguém que pretende receber um cheque fraudulentamente. Para evitar que uma quantia monetária seja alterada, a maioria dos sistemas computadorizados de preenchimento de cheque emprega uma técnica chamada *proteção de cheque*.

Cheques projetados para imprimir por computador contém um número fixo de espaços em que o computador pode imprimir uma quantia. Suponha que um cheque de pagamento contenha oito espaços em branco em que o computador deve imprimir a quantidade de um cheque de pagamento semanal. Se a quantidade for grande, então todos os oito espaços serão preenchidos, por exemplo:

1,230.60 (valor do cheque)

 12345678 (números de posição)

Por outro lado, se a quantidade for menor que \$ 1000, então vários dos espaços seriam comumente deixados em branco. Por exemplo,


```

99.87
-----
12345678

```

contém três espaços em branco. Se um cheque é impresso com espaços em branco, é mais fácil para alguém alterar o valor do cheque. Para evitar que um cheque seja alterado, muitos sistemas de preenchimento de cheque inserem *asteriscos iniciais* para proteger o valor como segue:

```

***99.87
-----
12345678

```

Escreva um programa que aceita como entrada uma quantia monetária para ser impressa em um cheque e então imprime o valor em formato de cheque protegido com asteriscos iniciais se necessário. Suponha que nove espaços estão disponíveis para imprimir uma quantia.

- 8.45** (*Valor de um cheque por extenso*) Continuando a discussão do exemplo anterior, reiteramos a importância de projetar sistemas de preenchimento de cheque para evitar alteração de seus valores. Um método comum de segurança requer que o valor do cheque seja escrito tanto em números como ‘por extenso’. Mesmo se alguém for capaz de alterar o valor numérico do cheque, é extremamente difícil alterar o valor por extenso.

Escreva um programa que aceita como entrada o valor do cheque em números e gera como saída o valor por extenso correspondente. Seu programa deve ser capaz de tratar valores de cheques tão altos quanto \$ 99,99. Por exemplo, o valor 112,43 deve ser escrito assim

```
ONE HUNDRED TWELVE and 43/100
```

- 8.46** (*Código Morse*) Talvez o mais famoso de todos os esquemas de codificação seja o código Morse, desenvolvido por Samuel Morse em 1832 para utilização com o sistema de telégrafo. O código Morse atribui uma série de pontos e traços a cada letra do alfabeto, a cada dígito e a alguns caracteres especiais (como ponto, vírgula, dois-pontos e ponto-e-vírgula). Em sistemas voltados para áudio, o ponto representa um som breve e o traço representa um som longo. Outras representações de pontos e traços são utilizadas com sistemas baseados em sinais luminosos e sistemas baseados em sinais de bandeira.

A separação entre palavras é indicada por um espaço ou, simplesmente, pela ausência de um ponto ou traço. Em um sistema baseado em áudio, um espaço é indicado por um ponto breve de tempo durante o qual nenhum som é transmitido. A versão internacional do código Morse aparece na Figura 8.45.

Escreva um programa que lê uma frase em inglês e a codifica em código Morse. Escreva também um programa que lê uma frase em código Morse e a converte no equivalente em inglês. Utilize um espaço em branco entre cada letra codificada em Morse e três espaços em branco entre cada palavra codificada em Morse.

Caractere	Código	Caractere	Código	Dígito	Código
A	.-	N	-.	1	.----
B	-...	O	---	2	..---
C	-.-.	P	.-..	3	...--
D	-..	Q	--.-	4	-
E	.	R	.-.	5	
F	..-.	S	...	6	-.....
G	--.	T	-	7	--....
H		U	..-	8	---..
I	..	V	...-	9	-----.
J	.----	W	.-..	0	-----
K	-.-	X	-.-.		
L	.-...	Y	-.--		
M	--	Z	--..		

Figura 8.45 Alfabeto do código Morse.

- 8.47** (*Um programa de conversão métrica*) Escreva um programa que ajudará o usuário a realizar conversões métricas. Seu programa deve permitir que o usuário especifique o nome das unidades como strings (isto é, centímetros, litros, gramas etc. para o sistema métrico e polegadas, quartos, libras etc. para o sistema inglês) e deve responder a perguntas simples como

"Quantas polegadas temos em 2 metros?"

"Quantos litros temos em 10 quartos?"

Seu programa deve reconhecer conversões inválidas. Por exemplo, a pergunta

"Quantos pés temos em 5 quilos?"

não é significativo, porque "pés" são unidades de comprimento, enquanto "quilos" são unidades de peso.

Um projeto desafiador de manipulação de string

- 8.48** (*Um gerador de palavras cruzadas*) A maioria das pessoas já brincou de palavras cruzadas, mas poucos tentaram gerar um jogo de palavras cruzadas. Gerar um programa de palavras cruzadas é um problema difícil. Isso é sugerido aqui como um projeto de manipulação de string que exige bastante sofisticação e esforço. Há muitas questões que o programador deve resolver para fazer funcionar até mesmo o mais simples programa gerador de palavras cruzadas. Por exemplo, como se representa a grade das palavras cruzadas dentro do computador? Você deve utilizar uma série de strings ou arrays bidimensionais? O programador precisa de uma fonte de palavras (isto é, um dicionário computadorizado) que possa ser referenciado diretamente pelo programa. De que forma essas palavras devem ser armazenadas para facilitar as complexas manipulações requeridas pelo programa? O leitor realmente ambicioso vai querer gerar a parte das 'pistas' do jogo, em que as breves dicas para as palavras 'horizontais' e para as palavras 'verticais' são impressas para o gerador de quebra-cabeça. A mera impressão de uma versão da parte em branco do jogo não é um problema simples.



*Meu desígnio, todo sublime,
a tempo hei de alcançar.*
W. S. Gilbert

*Este é um mundo no qual
devemos esconder virtudes?*
William Shakespeare

*Não seja 'consistente', mas
simplesmente autêntico.*
Oliver Wendell Holmes, Jr.

*Mas, sobretudo, sê a ti próprio
fiel.*
William Shakespeare

Classes: um exame mais profundo, parte I

OBJETIVOS

Neste capítulo, você aprenderá:

- A utilizar um empacotador de pré-processador para evitar múltiplos erros de definição causados pela inclusão de mais de uma cópia de um arquivo de cabeçalho em um arquivo de código-fonte.
- A definir o escopo de classe e acessar membros de classe via o nome de um objeto, uma referência a um objeto ou um ponteiro para um objeto.
- A definir construtores com argumentos-padrão.
- Como os destrutores são utilizados para realizar uma 'faxina de terminação' em um objeto antes de ele ser destruído.
- Quando construtores e destrutores são chamados e a ordem em que são chamados.
- Os erros de lógica que podem ocorrer quando uma função-membro `public` de uma classe retorna uma referência a dados `private`.
- A atribuir os membros de dados de um objeto àqueles de outro objeto por atribuição-padrão de membro a membro.

- 9.1 Introdução
- 9.2 Estudo de caso da classe `Time`
- 9.3 Escopo de classe e acesso a membros de classe
- 9.4 Separando a interface da implementação
- 9.5 Funções de acesso e funções utilitárias
- 9.6 Estudo de caso da classe `Time`: construtores com argumentos-padrão
- 9.7 Destrutores
- 9.8 Quando construtores e destrutores são chamados
- 9.9 Estudo de caso da classe `Time`: uma armadilha sutil — retornar uma referência a um membro de dados `private`
- 9.10 Atribuição-padrão de membro a membro
- 9.11 Reusabilidade de software
- 9.12 Estudo de caso de engenharia de software: começando a programar as classes do sistema ATM (opcional)
- 9.13 Síntese

Resumo | Terminologia | Exercícios de revisão | Respostas dos exercícios de revisão | Exercícios

9.1 Introdução

Nos capítulos anteriores, introduzimos muitos termos e conceitos básicos da programação C++ orientada a objetos. Além disso, discutimos nossa metodologia de desenvolvimento de programa: selecionamos atributos e comportamentos apropriados para cada classe e especificamos a maneira como os objetos de nossas classes colaboravam com os objetos de classes da C++ Standard Library para realizar os objetivos gerais de cada programa.

Neste capítulo, fazemos um exame mais profundo das classes. Utilizamos um estudo de caso integrado da classe `Time` neste capítulo (três exemplos) e no Capítulo 10 (dois exemplos) para demonstrar vários recursos da construção de classes. Iniciamos com uma classe `Time` que revisa vários dos recursos apresentados nos capítulos anteriores. O exemplo também demonstra um importante conceito de engenharia de software do C++ — utilizar um ‘empacotador de pré-processador’ em arquivos de cabeçalho para impedir que o código no cabeçalho seja incluído no mesmo arquivo de código-fonte mais de uma vez. Visto que uma classe pode ser definida apenas uma vez, utilizar essas diretivas de pré-processador impede erros de múltiplas definições.

Em seguida, discutimos o escopo de classe e os relacionamentos entre membros de uma classe. Demonstramos também como o código-cliente pode acessar membros `public` de uma classe por meio de três tipos de ‘handles’ — o nome de um objeto, uma referência a um objeto ou um ponteiro para um objeto. Como você verá, os nomes e as referências do objeto podem ser utilizados com o operador de seleção de membro ponto (.) para acessar um membro `public`, e os ponteiros podem ser utilizados com o operador de seleção de membro seta (->).

Discutimos funções de acesso que podem ler ou exibir dados em um objeto. Uma utilização comum de funções de acesso é testar a verdade ou falsidade de condições — essas funções são conhecidas como funções predicado. Também demonstramos a noção de uma função utilitária (também chamada função auxiliar) — uma função-membro `private` que suporta a operação de funções-membro `public` da classe, mas não é projetada para ser utilizada por clientes da classe.

No segundo exemplo do estudo de caso da classe `Time` demonstramos como passar argumentos para construtores e de que maneira argumentos-padrão podem ser utilizados no construtor para permitir que o código-cliente inicialize objetos de uma classe utilizando uma variedade de argumentos. Em seguida, discutimos uma função-membro especial chamada destrutor que faz parte de cada classe e é utilizada para realizar uma ‘faxina de terminação’ em um objeto antes de ele ser destruído. Então demonstramos a ordem em que construtores e destrutores são chamados, porque a correteza dos seus programas depende do uso adequado de objetos inicializados que ainda não foram destruídos.

Nosso último exemplo do estudo de caso da classe `Time` neste capítulo mostra uma prática de programação perigosa em que uma função-membro retorna uma referência a dados `private`. Discutimos como isso quebra o encapsulamento de uma classe e permite que o código-cliente acesse diretamente os dados de um objeto. Esse último exemplo mostra que os objetos da mesma classe podem ser atribuídos uns aos outros utilizando a atribuição-padrão de membro a membro, a qual copia os membros de dados no objeto à direita da atribuição para os membros de dados correspondentes do objeto à esquerda da atribuição. O capítulo conclui com uma discussão sobre a reusabilidade de software.

9.2 Estudo de caso da classe `Time`

Nosso primeiro exemplo (figuras 9.1–9.3) cria a classe `Time` e um programa de *test-drive* que testa a classe. Você já criou várias classes neste livro. Nesta seção, revisamos muitos dos conceitos tratados no Capítulo 3 e demonstramos um importante conceito da engenharia

de software do C++ — utilizar um ‘empacotador de pré-processador’ em arquivos de cabeçalho para impedir que o código no cabeçalho seja incluído no mesmo arquivo de código-fonte mais de uma vez. Visto que uma classe só pode ser definida uma única vez, utilizar essas diretivas de pré-processador impede erros de múltiplas definições.

Definição da classe `Time`

A definição de classe (Figura 9.1) contém protótipos (linhas 13–16) para as funções-membro `Time`, `setTime`, `printUniversal` e `printStandard`. A classe inclui os membros de inteiro `private` `hour`, `minute` e `second` (linhas 18–20). Os membros de dados `private` da classe `Time` podem ser acessados somente pelas suas quatro funções-membro. O Capítulo 12 introduz um terceiro especificador de acesso, `protected`, já que estudamos herança e o papel que ela desempenha na programação orientada a objetos.



Boa prática de programação 9.1

Por questão de clareza e legibilidade, utilize cada especificador de acesso uma única vez em uma definição de classe. Coloque os membros `public` primeiro, onde eles são fáceis de encontrar.



Observação de engenharia de software 9.1

Todo elemento de uma classe deve ter a visibilidade `private` a menos que possa ser comprovado que o elemento precise de visibilidade `public`. Esse é outro exemplo do princípio do menor privilégio.

Na Figura 9.1, observe que a definição de classe está incluída no seguinte **empacotador de pré-processador** (linhas 5–7 e 23):

```
// impede múltiplas inclusões de arquivo de cabeçalho
#ifndef TIME_H
#define TIME_H
...
#endif
```

Quando construirmos programas maiores, outras definições e declarações também serão colocadas em arquivos de cabeçalho. O empacotador de pré-processador anterior impede que o código entre `#ifndef` (que quer dizer ‘se não definido’) e `#endif` seja incluído se o nome `TIME_H` tiver sido definido. Se o cabeçalho não foi incluído anteriormente em um arquivo, o nome `TIME_H` será definido pela diretiva `#define` e as instruções de arquivo de cabeçalho serão incluídas. Se o cabeçalho tiver sido incluído, `TIME_H` já estará definido e o arquivo de cabeçalho não será novamente incluído. Tentativas de incluir um arquivo de cabeçalho múltiplas vezes (inadvertidamente) em

```
1 // Figura 9.1: Time.h
2 // Declaração da classe Time.
3 // Funções-membro são definidas em Time.cpp
4
5 // impede múltiplas inclusões de arquivo de cabeçalho
6 #ifndef TIME_H
7 #define TIME_H
8
9 // definição da classe Time
10 class Time
11 {
12 public:
13     Time(); // construtor
14     void setTime( int, int, int ); // configura hora, minuto e segundo
15     void printUniversal(); // imprime a hora no formato de data/hora universal
16     void printStandard(); // imprime a hora no formato-padrão de data/hora
17 private:
18     int hour; // 0 - 23 (formato de relógio de 24 horas)
19     int minute; // 0 - 59
20     int second; // 0 - 59
21 }; // fim da classe Time
22
23 #endif
```

Figura 9.1 Definição da classe `Time`.

geral ocorrem em programas grandes com muitos arquivos de cabeçalho que podem eles mesmos incluir outros arquivos de cabeçalho. [Nota: A convenção comumente utilizada para o nome da constante simbólica nas diretivas de pré-processador é simplesmente o nome de arquivo de cabeçalho em maiúscula com o caractere sublinhado substituindo o ponto.]



Dica de prevenção de erro 9.1

Utilize diretivas de pré-processador `#ifndef`, `#define` e `#endif` para formar um empacotador de pré-processador que impede que os arquivos de cabeçalho sejam incluídos mais de uma vez em um programa.



Boa prática de programação 9.2

Utilize o nome do arquivo do cabeçalho em caixa alta com o ponto substituído por um sublinhado nas diretivas de pré-processador `#ifndef` e `#define` de um arquivo de cabeçalho.

Funções-membro da classe *Time*

Na Figura 9.2, o construtor de *Time* (linhas 14–17) inicializa os membros de dados como 0 (isto é, o equivalente de data/hora universal de 12 AM). Isso assegura que o objeto inicia em um estado consistente. Os valores inválidos não podem ser armazenados nos membros de dados de um objeto *Time*, porque o construtor é chamado quando o objeto *Time* é criado e todas as tentativas subsequentes de um cliente de modificar os membros de dados serão escrutinadas pela função `setTime` (discutida em breve). Por fim, é importante observar que o programador pode definir vários construtores sobrecarregados para uma classe.

Os membros de dados de uma classe não podem ser inicializados onde são declarados, no corpo da classe. É altamente recomendado que esses membros de dados sejam inicializados pelo construtor da classe (como não há inicialização-padrão para membros de dados de tipo fundamental). Os membros de dados também podem receber valores pelas funções `set` de *Time*. [Nota: O Capítulo 10 demonstra que apenas os membros de dados `static const` de uma classe dos tipos inteiro ou `enum` podem ser inicializados no corpo da classe.]



Erro comum de programação 9.1

Tentar inicializar explicitamente um membro de dados não-`static` de uma classe na definição de classe é um erro de sintaxe.

A função `setTime` (linhas 21–26) é uma função `public` que declara três parâmetros `int` e os utiliza para configurar a hora. Uma expressão condicional testa cada argumento para determinar se o valor está dentro de um intervalo especificado. Por exemplo, o valor `hour` (linha 23) deve ser maior ou igual a 0 e menor que 24, porque o formato de data/hora universal representa horas como inteiros de 0 a 23 (por exemplo, 1 PM é a hora 13 e 11 PM é a hora 23; meia-noite é a hora 0 e meio-dia é a hora 12). De maneira semelhante, tanto os valores `minute` como `second` (linhas 24 e 25) devem ser maiores ou iguais a 0 e menores que 60. Quaisquer valores fora desses intervalos são configurados como zero para assegurar que um objeto *Time* sempre contenha dados consistentes — isto é, os valores dos dados do objeto sempre são mantidos em um intervalo, mesmo se os valores fornecidos como argumentos para a função `setTime` estiverem incorretos. Nesse exemplo, zero é um valor consistente para `hour`, `minute` e `second`.

Um valor passado para `setTime` é um valor correto se estiver no intervalo permitido para o membro que ele está inicializando. Portanto, qualquer número no intervalo de 0 a 23 seria um valor correto para `hour`. Um valor correto sempre é um valor consistente. Entretanto, um valor consistente não necessariamente é um valor correto. Se `setTime` configurar `hour` como 0 porque o argumento recebido estava fora do intervalo, então `hour` será correto somente se a hora atual for, coincidentemente, meia-noite.

A função `printUniversal` (linhas 29–33 da Figura 9.2) não aceita argumentos e gera saída da data no formato de data/hora universal, consistindo em três pares de dígitos separados por dois-pontos — para a hora, minuto e segundo, respectivamente. Por exemplo, se a data/hora fosse 1:30:07 PM, a função `printUniversal` retornaria 13:30:07. Observe que a linha 31 utiliza o manipulador de fluxo parametrizado `setfill` para especificar o **caractere de preenchimento** que é exibido quando um inteiro é enviado para a saída em um campo maior do que o número de dígitos no valor. Por padrão, os caracteres de preenchimento aparecem à esquerda dos dígitos no número. Nesse exemplo, se o valor `minute` for 2, ele será exibido como 02, porque o caractere de preenchimento está configurado com zero ('0'). Se o número cuja saída está sendo gerada preencher o campo especificado, o caractere de preenchimento não será exibido. Observe que, uma vez que o caractere de preenchimento é especificado com `setfill`, ele se aplica a todos valores subsequentes que são exibidos em campos maiores do que o valor sendo exibido (isto é, `setfill` é uma configuração 'aderente'). Isso contrasta com `setw`, que se aplica somente ao próximo valor exibido (`setw` é uma configuração 'não aderente').



Dica de prevenção de erro 9.2

Toda configuração aderente (como um caractere de preenchimento ou a precisão de ponto flutuante) deve ser restaurada à sua configuração anterior quando não for mais necessária. A falha em fazer isso pode resultar em uma saída formatada incorretamente mais tarde em um programa. O Capítulo 15, “Entrada/saída de fluxo”, discute como redefinir o caractere de preenchimento e a precisão.

A função `printStandard` (linhas 36–41) não recebe argumentos e gera saída de uma data no formato-padrão de data/hora, que consiste nos valores `hour`, `minute` e `second` separados por dois-pontos e seguidos por um indicador AM ou PM (por exemplo, 1:27:06 PM).

```

1 // Figura 9.2: Time.cpp
2 // Definições de função-membro para a classe Time.
3 #include <iostream>
4 using std::cout;
5
6 #include <iomanip>
7 using std::setfill;
8 using std::setw;
9
10 #include "Time.h" // inclui a definição da classe Time a partir de Time.h
11
12 // O construtor de Time inicializa cada membro de dados como zero.
13 // Assegura que todos os objetos Time iniciem em um estado consistente.
14 Time::Time()
15 {
16     hour = minute = second = 0;
17 } // fim do construtor de Time
18
19 // configura novo valor de Time utilizando a hora universal; assegura que
20 // os dados permaneçam consistentes configurando valores inválidos como zero
21 void Time::setTime( int h, int m, int s )
22 {
23     hour = ( h >= 0 && h < 24 ) ? h : 0; // valida horas
24     minute = ( m >= 0 && m < 60 ) ? m : 0; // valida minutos
25     second = ( s >= 0 && s < 60 ) ? s : 0; // valida segundos
26 } // fim da função setTime
27
28 // imprime a hora no formato de data/hora universal (HH:MM:SS)
29 void Time::printUniversal()
30 {
31     cout << setfill( '0' ) << setw( 2 ) << hour << ":"
32         << setw( 2 ) << minute << ":" << setw( 2 ) << second;
33 } // fim da função printUniversal
34
35 // imprime a hora no formato-padrão de data/hora (HH:MM:SS AM ou PM)
36 void Time::printStandard()
37 {
38     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 ) << ":"
39         << setfill( '0' ) << setw( 2 ) << minute << ":" << setw( 2 )
40         << second << ( hour < 12 ? " AM" : " PM" );
41 } // fim da função printStandard

```

Figura 9.2 Definições de função-membro da classe Time.

Semelhantemente à função `printUniversal`, a função `printStandard` utiliza `setfill('0')` para formatar `minute` e `second` como dois valores de dígito com zeros à esquerda se necessário. A linha 38 utiliza um operador condicional (`?:`) para determinar o valor de `hour` a ser exibido — se `hour` for 0 ou 12 (AM ou PM), ela aparece como 12; caso contrário, `hour` aparece como um valor de 1 a 11. O operador condicional na linha 40 determina se AM ou PM será exibido.

Definindo funções-membro fora da definição de classe; escopo de classe

Embora uma função-membro declarada em uma definição de classe possa ser definida fora dessa definição de classe (e associada à classe via operador de resolução de escopo binário), essa função-membro ainda está dentro desse **escopo de classe**; isto é, seu nome é conhecido apenas por outros membros da classe a menos que referenciado via um objeto da classe, uma referência a um objeto da classe, um ponteiro para um objeto da classe ou o operador binário de resolução de escopo. Em breve, falaremos mais sobre o escopo de classe.

Se uma função-membro é definida no corpo de uma definição de classe, o compilador C++ tenta colocar inline as chamadas para função-membro. As funções-membro definidas fora de uma definição de classe podem ser colocadas inline utilizando a palavra-chave `inline` explicitamente. Lembre-se de que o compilador reserva-se o direito de não colocar inline nenhuma função.

**Dica de desempenho 9.1**

Definir uma função-membro dentro da definição de classe coloca a função-membro inline (se o compilador escolher fazer isso). Isso pode melhorar o desempenho.

**Observação de engenharia de software 9.2**

Definir uma pequena função-membro dentro da definição de classe não promove a melhor engenharia de software, porque os clientes da classe serão capazes de ver a implementação da função, e o código-cliente deve ser recompilado se a definição de função mudar.

**Observação de engenharia de software 9.3**

Apenas as funções-membro mais simples e mais estáveis (isto é, cujas implementações provavelmente não mudarão) devem ser definidas no cabeçalho de classe.

Funções-membro versus funções globais

É interessante que as funções-membro `printUniversal` e `printStandard` não aceitem nenhum argumento. Isso ocorre porque essas funções-membro sabem implicitamente que devem imprimir os membros de dados do objeto `Time` particular para o qual elas são invocadas. Isso pode tornar as chamadas de função-membro mais concisas do que as chamadas de função convencionais na programação procedural.

**Observação de engenharia de software 9.4**

Utilizar freqüentemente uma abordagem de programação orientada a objetos pode simplificar chamadas de função reduzindo o número de parâmetros que será passado. Esse benefício da programação orientada a objetos deriva do fato de que encapsular os membros de dados e as funções-membro dentro de um objeto dá às funções-membro o direito de acessar os membros de dados.

**Observação de engenharia de software 9.5**

Em geral, as funções-membro são mais curtas do que funções em programas não orientados a objeto, porque os dados armazenados em membros de dados foram idealmente validados por um construtor ou por funções-membro que armazenam novos dados. Como os dados já estão no objeto, as chamadas de função-membro freqüentemente não têm argumentos ou pelo menos têm menos argumentos que as típicas chamadas de função em linguagens não orientadas a objeto. Portanto, as chamadas, as definições de função e os protótipos de função são menores. Isso facilita muitos aspectos do desenvolvimento de programa.

**Dica de prevenção de erro 9.3**

O fato de que as chamadas de função-membro geralmente não aceitam argumentos ou aceitam substancialmente menos argumentos do que as chamadas de função convencionais em linguagens não orientadas a objeto reduz a probabilidade de passar os argumentos errados, os tipos de argumentos errados ou o número errado de argumentos.

Utilizando a classe `Time`

Uma vez que a classe `Time` foi definida, ela pode ser utilizada como um tipo em declarações de objeto, array, ponteiro e referência como mostrado a seguir:

```
Time sunset; // objeto do tipo Time
Time arrayOfTimes[ 5 ], // array de 5 objetos Time
Time &dinnerTime = sunset; // referência a um objeto Time
Time *timePtr = &dinnerTime, // ponteiro para um objeto Time
```

A Figura 9.3 utiliza a classe `Time`. A linha 12 instancia um único objeto da classe `Time` chamado `t`. Quando o objeto é instanciado, o construtor de `Time` é chamado para inicializar cada membro de dados `private` como 0. Então, as linhas 16 e 18 imprimem a hora nos formatos universal e padrão para confirmar que os membros foram inicializados adequadamente. A linha 20 configura uma nova hora chamando a função-membro `setTime` e as linhas 24 e 26 imprimem novamente a hora em ambos os formatos. A linha 28 tenta utilizar `setTime` para configurar os membros de dados como valores inválidos — a função `setTime` reconhece isso e configura os valores inválidos como 0 para manter o objeto em um estado consistente. Por fim, as linhas 33 e 35 imprimem novamente a hora em ambos os formatos.

```

1 // Figura 9.3: fig09_03.cpp
2 // Programa para testar a classe Time.
3 // NOTA: Esse arquivo deve ser compilado com Time.cpp.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include "Time.h" // inclui a definição da classe Time a partir de Time.h
9
10 int main()
11 {
12     Time t; // instancia o objeto t da classe Time
13
14     // gera saída de valores iniciais do objeto Time t
15     cout << "The initial universal time is ";
16     t.printUniversal(); // 00:00:00
17     cout << "\nThe initial standard time is ";
18     t.printStandard(); // 12:00:00 AM
19
20     t.setTime( 13, 27, 6 ); // muda a hora
21
22     // gera saída de novos valores do objeto Time t
23     cout << "\n\nUniversal time after setTime is ";
24     t.printUniversal(); // 13:27:06
25     cout << "\nStandard time after setTime is ";
26     t.printStandard(); // 1:27:06 PM
27
28     t.setTime( 99, 99, 99 ); // tenta configurações inválidas
29
30     // gera saída de valores de t depois de especificar valores inválidos
31     cout << "\n\nAfter attempting invalid settings:"
32         << "\nUniversal time: ";
33     t.printUniversal(); // 00:00:00
34     cout << "\nStandard time: ";
35     t.printStandard(); // 12:00:00 AM
36     cout << endl;
37     return 0;
38 } // fim de main

```

```

The initial universal time is 00:00:00
The initial standard time is 12:00:00 AM

Universal time after setTime is 13:27:06
Standard time after setTime is 1:27:06 PM

After attempting invalid settings:
Universal time: 00:00:00
Standard time: 12:00:00 AM

```

Figura 9.3 Programa para testar a classe Time.

Planejando-se para composição e herança

Freqüentemente, as classes não precisam ser criadas ‘a partir do zero’. Em vez disso, elas podem incluir objetos de outras classes como membros ou podem ser **derivadas** de outras classes que fornecem atributos e comportamentos que as novas classes podem utilizar. Essa reutilização de software pode aprimorar significativamente a produtividade do programador e simplificar a manutenção de código. A inclusão de objetos de uma classe como membros de outras classes é chamada de **composição** (ou **agregação**) e é discutida no Capítulo 10. A derivação de novas classes de classes existentes é chamada de **herança** e é discutida no Capítulo 12.

Tamanho de objeto

As pessoas sem experiência em programação orientada a objetos frequentemente supõem que os objetos devem ser bem grandes porque eles contêm membros de dados e funções-membro. Logicamente, isso é verdade — o programador pode pensar nos objetos como contendo dados e funções (e nossa discussão certamente encorajou essa visão); fisicamente, porém, isso não é verdade.



Dica de desempenho 9.2

*Os objetos contêm apenas dados, portanto, os objetos são muito menores do que se também contivessem funções-membro. Aplicar o operador `sizeof` a um nome de classe ou a um objeto dessa classe informará somente o tamanho dos membros de dados da classe. O compilador cria uma (única) cópia das funções-membro separada de todos os objetos da classe. Todos os objetos da classe compartilham esta única cópia. Cada objeto precisa, naturalmente, de sua própria cópia dos dados da classe, porque os dados podem variar entre os objetos. O código de função é não modificável (também chamado de **código reentrante** ou **procedure pura**) e, daí, pode ser compartilhado entre todos os objetos de uma classe.*

9.3 Escopo de classe e acesso a membros de classe

Os membros de dados de uma classe (variáveis declaradas na definição de classe) e as funções-membro (funções declaradas na definição de classe) pertencem ao escopo dessa classe. As funções não-membro são definidas no **escopo de arquivo**.

Dentro do escopo de uma classe, os membros de classe são imediatamente acessíveis por todas as funções-membro dessa classe e podem ser referenciados por nome. Fora do escopo de uma classe, os membros de classe `public` são referenciados por um dos **handles** em um objeto — um nome de objeto, uma referência a um objeto ou um ponteiro para um objeto. O tipo do objeto, referência ou ponteiro especifica a interface (isto é, as funções-membro) acessível(is) ao cliente. [Veremos no Capítulo 10 que um handle implícito é inserido pelo compilador em cada referência a um membro de dados ou função-membro a partir de dentro de um objeto.]

As funções-membro de uma classe podem ser sobrecarregadas, mas apenas por outras funções-membro dessa classe. Para sobrecarregar uma função-membro, simplesmente forneça na definição de classe um protótipo para cada versão da função sobrecarregada e uma definição de função separada para cada versão da função.

As variáveis declaradas em uma função-membro têm escopo de bloco e são conhecidas somente por essa função. Se uma função-membro definir uma variável com o mesmo nome de uma variável com escopo de classe, a variável de escopo de classe é ocultada pela variável de escopo de bloco no escopo de bloco. Essa variável oculta pode ser acessada colocando-se o nome da classe seguido pelo operador de resolução de escopo (`::`) antes do nome da variável. As variáveis globais ocultas podem ser acessadas com o operador unário de resolução de escopo (ver o Capítulo 6).

O operador de seleção de membro ponto (`.`) é precedido pelo nome de um objeto ou com uma referência a um objeto para acessar os membros do objeto. O operador de seleção de membro seta (`->`) é precedido por um ponteiro para um objeto para acessar os membros do objeto.

A Figura 9.4 utiliza uma classe simples chamada `Count` (linhas 8–25) com o membro de dados `private x` do tipo `int` (linha 24), a função-membro `public setX` (linhas 12–15) e a função-membro `public print` (linhas 18–21), para ilustrar o acesso aos membros de uma classe com os operadores de seleção de membro. Para simplificar, incluímos essa pequena classe no mesmo arquivo da função `main` que a utiliza. As linhas 29–31 criam três variáveis relacionadas com o tipo `Count` — `counter` (um objeto `Count`), `counterPtr` (um ponteiro para um objeto `Count`) e `counterRef` (uma referência a um objeto `Count`). A variável `counterRef` referencia `counter` e a variável `counterPtr` aponta para `counter`. Nas linhas 34–35 e 38–39, observe que o programa pode invocar funções-membro `setX` e `print` utilizando o operador de seleção de membro ponto (`.`) precedido pelo nome do objeto (`counter`) ou por uma referência ao objeto (`counterRef`, que é um alias para `counter`). De maneira semelhante, as linhas 42–43 demonstram que o programa pode invocar funções-membro `setX` e `print` utilizando um ponteiro (`countPtr`) e o operador de seleção de membro seta (`->`).

9.4 Separando a interface da implementação

No Capítulo 3 começamos incluindo a definição de uma classe e definições de função-membro em um arquivo. Em seguida, demonstramos a separação desse código em dois arquivos — um arquivo de cabeçalho para a definição de classe (isto é, a interface da classe) e um arquivo de código-fonte para as definições de função-membro da classe (isto é, a implementação da classe). Lembre-se de que isso facilita a modificação de programas — contanto que os clientes de uma classe estejam envolvidos, as alterações na implementação da classe não afetam o cliente desde que a interface da classe originalmente fornecida ao cliente permaneça inalterada.



Observação de engenharia de software 9.6

Os clientes de uma classe não precisam de acesso ao código-fonte da classe para utilizá-la. Entretanto, os clientes precisam, de fato, ser capazes de se vincular ao código-objeto da classe (isto é, a versão compilada da classe). Isso encoraja os fornecedores de softwares independentes (independent software vendors – ISVs) a comercializar ou licenciar bibliotecas de classes. Os ISVs fornecem em seus produtos somente os arquivos de cabeçalho e os módulos dos objetos. Informações ‘não-proprietárias’ (isto é, não patenteadas) são reveladas — como seria o caso se o código-fonte fosse fornecido. A comunidade de usuários do C++ se beneficia tendo mais bibliotecas de classes produzidas e disponibilizadas por ISVs.

```

1 // Figura 9.4: fig09_04.cpp
2 // Demonstrando os operadores de acesso ao membro de classe com . e ->
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 // definição da classe Count
8 class Count
9 {
10 public: // dados public são perigosos
11     // configura o valor do membro de dados private x
12     void setX( int value )
13     {
14         x = value;
15     } // fim da função setX
16
17     // imprime o valor do membro de dados private x
18     void print()
19     {
20         cout << x << endl;
21     } // fim da função print
22
23 private:
24     int x;
25 }; // fim da classe Count
26
27 int main()
28 {
29     Count counter; // cria objeto counter
30     Count *counterPtr = &counter; // cria ponteiro para counter
31     Count &counterRef = counter; // criar referência para counter
32
33     cout << "Set x to 1 and print using the object's name: ";
34     counter.setX( 1 ); // configura membro de dados x como 1
35     counter.print(); // chama função-membro print
36
37     cout << "Set x to 2 and print using a reference to an object: ";
38     counterRef.setX( 2 ); // configura membro de dados x como 2
39     counterRef.print(); // chama função-membro print
40
41     cout << "Set x to 3 and print using a pointer to an object: ";
42     counterPtr->setX( 3 ); // configura membro de dados x como 3
43     counterPtr->print(); // chama função-membro print
44     return 0;
45 } // fim de main

```

```

Set x to 1 and print using the object's name: 1
Set x to 2 and print using a reference to an object: 2
Set x to 3 and print using a pointer to an object: 3

```

Figura 9.4 Acessando funções-membro de um objeto por meio de cada tipo de handle de objeto — o nome do objeto, uma referência ao objeto e um ponteiro para o objeto.

Na realidade, as coisas não são essa maravilha toda. Os arquivos de cabeçalho contêm algumas partes da implementação e dicas sobre outras. As funções-membro inline, por exemplo, precisam estar em um arquivo de cabeçalho para que, quando o compilador compilar um cliente, este possa incluir adequadamente a definição de função `inline`. Os membros `private` de uma classe são listados na definição de classe no arquivo de cabeçalho, portanto esses membros são visíveis aos clientes mesmo que os clientes possam não acessar os membros `private`. No Capítulo 10, mostramos como utilizar uma ‘classe proxy’ para ocultar até mesmo os dados `private` de uma classe a partir de clientes da classe.



Observação de engenharia de software 9.7

As informações importantes para a interface para uma classe devem ser incluídas no arquivo de cabeçalho. As informações que só serão utilizadas internamente na classe e não serão necessárias aos clientes da classe devem ser incluídas no arquivo-fonte não publicado. Esse é ainda outro exemplo do princípio do menor privilégio.

9.5 Funções de acesso e funções utilitárias

As **funções de acesso** podem ler ou exibir dados. Outra utilização comum das funções de acesso é testar a verdade ou falsidade de condições — essas são freqüentemente chamadas de **funções predicado**. Um exemplo de uma função predicado seria uma função `isEmpty` para qualquer classe contêiner — uma classe capaz de armazenar muitos objetos — como uma lista vinculada, uma pilha ou uma fila. Um programa talvez teste `isEmpty` antes de tentar ler outro item do objeto contêiner. Uma função predicado `isFull` poderia testar um objeto de classe contêiner para determinar se ela tem ou não espaço adicional. As funções predicado úteis para nossa classe `Time` poderiam ser `isAM` e `isPM`.

O programa das figuras 9.5–9.7 demonstram a noção de uma função utilitária (também chamada de função **auxiliar**). Uma função utilitária não faz parte da interface pública de uma classe; em vez disso, é uma função-membro `private` que suporta a operação das funções-membro pública da classe. As funções utilitárias não são projetadas para ser utilizadas por clientes de uma classe (mas podem ser utilizadas por `friends` de uma classe, como veremos no Capítulo 10).

A classe `SalesPerson` (Figura 9.5) declara um array de 12 estimativas de vendas mensais (linha 16) e os protótipos para o construtor e as funções-membro da classe que manipulam o array.

Na Figura 9.6, o construtor `SalesPerson` (linhas 15–19) inicializa o array `sales` como zero. A função-membro pública `setSales` (linhas 36–43) configura a estimativa de vendas de um mês no array `sales`. A função-membro pública `printAnnualSales` (linhas 46–51) imprime o total de vendas dos últimos 12 meses. A função utilitária `private totalAnnualSales` (linhas 54–62) soma as 12 estimativas mensais de vendas em benefício de `printAnnualSales`. A função-membro `printAnnualSales` edita as estimativas de vendas em formato monetário.

Na Figura 9.7 note que a função `main` do aplicativo inclui uma única sequência simples de chamadas de função-membro — não há nenhuma instrução de controle. A lógica de manipular o array `sales` está completamente encapsulada nas funções-membro da classe `SalesPerson`.

```

1 // Figura 9.5: SalesPerson.h
2 // Definição da classe SalesPerson.
3 // Funções-membro definidas em SalesPerson.cpp.
4 #ifndef SALESP_H
5 #define SALESP_H
6
7 class SalesPerson
8 {
9 public:
10     SalesPerson(); // construtor
11     void getSalesFromUser(); // insere as vendas a partir do teclado
12     void setSales( int, double ); // configura as vendas de um mês específico
13     void printAnnualSales(); // resume e imprime as vendas
14 private:
15     double totalAnnualSales(); // protótipo para função utilitária
16     double sales[ 12 ]; // 12 estimativas de vendas mensais
17 }; // fim da classe SalesPerson
18
19 #endif

```

Figura 9.5 Definição da classe `SalesPerson`.

```

1 // Figura 9.6: SalesPerson.cpp
2 // Funções-membro para a classe SalesPerson.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7 using std::fixed;
8
9 #include <iomanip>
10 using std::setprecision;
11
12 #include "SalesPerson.h" // inclui definição da classe SalesPerson
13
14 // inicializa elementos do array sales como 0.0
15 SalesPerson::SalesPerson()
16 {
17     for ( int i = 0; i < 12; i++ )
18         sales[ i ] = 0.0;
19 } // fim do construtor SalesPerson
20
21 // obtém 12 estimativas de vendas do usuário no teclado
22 void SalesPerson::getSalesFromUser()
23 {
24     double salesFigure;
25
26     for ( int i = 1; i <= 12; i++ )
27     {
28         cout << "Enter sales amount for month " << i << ": ";
29         cin >> salesFigure;
30         setSales( i, salesFigure );
31     } // fim do for
32 } // fim da função getSalesFromUser
33
34 // configura uma das 12 estimativas de vendas mensais; a função subtrai
35 // um do valor mensal para o subscrito adequado no array sales
36 void SalesPerson::setSales( int month, double amount )
37 {
38     // testa a validade do mês e do valor
39     if ( month >= 1 && month <= 12 && amount > 0 )
40         sales[ month - 1 ] = amount; // ajusta para subscritos 0-11
41     else // mês ou valor inválido
42         cout << "Invalid month or sales figure" << endl;
43 } // fim da função setSales
44
45 // imprime o total das vendas anuais (com a ajuda da função utilitária)
46 void SalesPerson::printAnnualSales()
47 {
48     cout << setprecision( 2 ) << fixed
49         << "\nThe total annual sales are: $"
50         << totalAnnualSales() << endl; // chama a função utilitária
51 } // fim da função printAnnualSales
52
53 // função utilitária private para somar vendas anuais
54 double SalesPerson::totalAnnualSales()
55 {
56     double total = 0.0; // inicializa o total
57

```

Figura 9.6 Definições de função-membro da classe SalesPerson.

(continua)

```

58     for ( int i = 0; i < 12; i++ ) // resume resultados de vendas
59         total += sales[ i ]; // adiciona vendas do mês i ao total
60
61     return total;
62 } // fim da função totalAnnualSales

```

Figura 9.6 Definições de função-membro da classe SalesPerson.

(continuação)

```

1 // Figura 9.7: fig09_07.cpp
2 // Demonstrando uma função utilitária.
3 // Compile esse programa com SalesPerson.cpp
4
5 // inclui a definição da classe SalesPerson a partir de SalesPerson.h
6 #include "SalesPerson.h"
7
8 int main()
9 {
10     SalesPerson s; // cria o objeto SalesPerson s
11
12     s.getSalesFromUser(); // anota código seqüencial simples;
13     s.printAnnualSales(); // nenhuma instrução de controle em main
14     return 0;
15 } // fim de main

```

```

Enter sales amount for month 1: 5314.76
Enter sales amount for month 2: 4292.38
Enter sales amount for month 3: 4589.83
Enter sales amount for month 4: 5534.03
Enter sales amount for month 5: 4376.34
Enter sales amount for month 6: 5698.45
Enter sales amount for month 7: 4439.22
Enter sales amount for month 8: 5893.57
Enter sales amount for month 9: 4909.67
Enter sales amount for month 10: 5123.45
Enter sales amount for month 11: 4024.97
Enter sales amount for month 12: 5923.92

```

The total annual sales are: \$60120.59

Figura 9.7 Demonstração da função utilitária.



Observação de engenharia de software 9.8

Um fenômeno da programação orientada a objetos é que, uma vez que uma classe é definida, a criação e a manipulação de objetos dessa classe envolvem freqüentemente emitir apenas uma seqüência simples de chamadas de função-membro — poucas instruções de controle, se houver, são necessárias. Por contraste, é comum ter instruções de controle na implementação de funções-membro de uma classe.

9.6 Estudo de caso da classe Time: construtores com argumentos-padrão

O programa das figuras 9.8–9.10 aprimora a classe Time para demonstrar como os argumentos são implicitamente passados para um construtor. O construtor definido na Figura 9.2 inicializou hour, minute e second como 0 (isto é, meia-noite no horário universal). Como outras funções, os construtores podem especificar argumentos-padrão. A linha 13 da Figura 9.8 declara o construtor de Time para incluir argumentos-padrão, especificando um valor-padrão de zero para cada argumento passado para o construtor. Na Figura 9.9, as linhas 14–17 definem a nova versão do construtor de Time que recebe valores dos parâmetros hr, min e sec que serão utilizados para

inicializar os membros de dados `private hour, minute` e `second`, respectivamente. Observe que a classe `Time` fornece as funções *set* e *get* para cada membro de dados. O construtor de `Time` agora chama `setTime`, que chama as funções `setHour`, `setMinute` e `setSecond` para validar e atribuir valores aos membros de dados. Os argumentos-padrão para o construtor asseguram que, mesmo se nenhum valor for fornecido em uma chamada de construtor, o construtor ainda inicializará os membros de dados para manter o objeto `Time` em um estado consistente. Um construtor que assume os padrões para todos os seus argumentos também é um construtor-padrão — isto é, um construtor que pode ser invocado sem argumentos. Pode haver um máximo de um construtor-padrão por classe.

Na Figura 9.9, a linha 16 do construtor chama a função-membro `setTime` com os valores passados para o construtor (ou os valores-padrão). A função `setTime` chama `setHour` para assegurar que o valor fornecido para `hour` esteja no intervalo 0–23, em seguida chama `setMinute` e `setSecond` para assegurar que os valores de `minute` e `second` estejam cada um no intervalo 0–59. Se um valor estiver fora do intervalo, esse valor é configurado como zero (para assegurar que cada membro de dados permaneça em um estado consistente). No Capítulo 16, “Tratamento de exceções”, lançamos as exceções para informar ao usuário de que um valor está fora do intervalo, em vez de simplesmente atribuir um valor-padrão consistente.

Observe que o construtor de `Time` poderia ser escrito para incluir as mesmas instruções que a função-membro `setTime`, ou até mesmo as instruções individuais nas funções `setHour`, `setMinute` e `setSecond`. Chamar `setHour`, `setMinute` e `setSecond` a partir do construtor pode ser ligeiramente mais eficiente porque a chamada extra a `setTime` seria eliminada. De maneira semelhante, copiar o código das linhas 31, 37 e 43 no construtor eliminaria o overhead de chamar `setTime`, `setHour`, `setMinute` e `setSecond`. Codificar o construtor de `Time` ou a função-membro `setTime` como uma cópia do código nas linhas 31, 37 e 43 tornaria a manutenção dessa classe mais difícil. Se as implementações de `setHour`, `setMinute` e `setSecond` precisarem ser alteradas, a implementação de qualquer função-membro que duplica as linhas 31, 37 e 43 teria de ser alterada de maneira correspondente. Fazer o construtor de `Time` chamar `setTime` e fazer `setTime` chamar `setHour`, `setMinute` e `setSecond` permite limitar as alterações no código que valida `hour`, `minute` ou `second`.

```

1 // Figura 9.8: Time.h
2 // Declaração da classe Time.
3 // Funções-membro definidas em Time.cpp.
4
5 // impede múltiplas inclusões de arquivo de cabeçalho
6 #ifndef TIME_H
7 #define TIME_H
8
9 // Definição de tipo de dados abstrato Time
10 class Time
11 {
12 public:
13     Time( int = 0, int = 0, int = 0 ); // construtor-padrão
14
15     // funções set
16     void setTime( int, int, int ); // configura hour, minute, second
17     void setHour( int ); // configura hour (depois da validação)
18     void setMinute( int ); // configura minute (depois da validação)
19     void setSecond( int ); // configura second (depois da validação)
20
21     // funções get
22     int getHour(); // retorna hour
23     int getMinute(); // retorna minute
24     int getSecond(); // retorna second
25
26     void printUniversal(); // gera saída da hora no formato universal de data/hora
27     void printStandard(); // gera saída da hora no formato-padrão de data/hora
28 private:
29     int hour; // 0 - 23 (formato de relógio de 24 horas)
30     int minute; // 0 - 59
31     int second; // 0 - 59
32 }; // fim da classe Time
33
34 #endif

```

Figura 9.8 A classe `Time` contendo um construtor com argumentos-padrão.

com a função `set` correspondente. Isso reduz a probabilidade de erros ao alterar a implementação da classe. Além disso, o desempenho do construtor de `Time` e de `setTime` pode ser aprimorado declarando-os explicitamente como `inline` ou definindo-os na definição de classe (que torna a definição de função implicitamente `inline`).



Observação de engenharia de software 9.9

Se a função-membro de uma classe já fornecer toda ou parte da funcionalidade requerida por um construtor (ou por outra função-membro) da classe, chame essa função-membro a partir do construtor (ou de outra função-membro). Isso simplifica a manutenção do código e reduz a probabilidade de um erro se a implementação do código for modificada. Como regra geral, evite a repetição de código.



Observação de engenharia de software 9.10

Qualquer alteração nos valores de argumento-padrão de uma função exige que o código-cliente seja recompilado (para assegurar que o programa ainda funciona corretamente).

```

1 // Figura 9.9: Time.cpp
2 // Definições de função-membro para a classe Time.
3 #include <iostream>
4 using std::cout;
5
6 #include <iomanip>
7 using std::setfill;
8 using std::setw;
9
10 #include "Time.h" // inclui a definição da classe Time a partir de Time.h
11
12 // Construtor de Time inicializa cada membro de dados como zero;
13 // assegura que os objetos Time iniciem em um estado consistente
14 Time::Time( int hr, int min, int sec )
15 {
16     setTime( hr, min, sec ); // valida e configura time
17 } // fim do construtor de Time
18
19 // configura novo valor de Time utilizando a hora universal; assegura que
20 // os dados permaneçam consistentes configurando valores inválidos como zero
21 void Time::setTime( int h, int m, int s )
22 {
23     setHour( h ); // configura campo private hour
24     setMinute( m ); // configura campo private minute
25     setSecond( s ); // configura campo private second
26 } // fim da função setTime
27
28 // configura valor de hour
29 void Time::setHour( int h )
30 {
31     hour = ( h >= 0 && h < 24 ) ? h : 0; // valida horas
32 } // fim da função setHour
33
34 // configura valor de minute
35 void Time::setMinute( int m )
36 {
37     minute = ( m >= 0 && m < 60 ) ? m : 0; // valida minutos
38 } // fim da função setMinute
39

```

Figura 9.9 Definições de função-membro da classe `Time` incluindo um construtor que aceita argumentos.

(continua)

```

40 // configura valor de second
41 void Time::setSecond( int s )
42 {
43     second = ( s >= 0 && s < 60 ) ? s : 0; // valida segundos
44 } // fim da função setSecond
45
46 // retorna valor de hour
47 int Time::getHour()
48 {
49     return hour;
50 } // fim da função getHour
51
52 // retorna valor de minute
53 int Time::getMinute()
54 {
55     return minute;
56 } // fim da função getMinute
57
58 // retorna valor de second
59 int Time::getSecond()
60 {
61     return second;
62 } // fim da função getSecond
63
64 // imprime a hora no formato universal de data/hora (HH:MM:SS)
65 void Time::printUniversal()
66 {
67     cout << setfill( '0' ) << setw( 2 ) << getHour() << ":"
68         << setw( 2 ) << getMinute() << ":" << setw( 2 ) << getSecond();
69 } // fim da função printUniversal
70
71 // imprime a hora no formato-padrão de data/hora (HH:MM:SS AM ou PM)
72 void Time::printStandard()
73 {
74     cout << ( ( getHour() == 0 || getHour() == 12 ) ? 12 : getHour() % 12 )
75         << ":" << setfill( '0' ) << setw( 2 ) << getMinute()
76         << ":" << setw( 2 ) << getSecond() << ( hour < 12 ? " AM" : " PM" );
77 } // fim da função printStandard

```

Figura 9.9 Definições de função-membro da classe Time incluindo um construtor que aceita argumentos.

(continuação)

A função `main` na Figura 9.10 inicializa cinco objetos Time — um com os três argumentos convertidos para sua configuração-padrão na chamada de construtor implícita (linha 11), um com um argumento especificado (linha 12), um com dois argumentos especificados (linha 13), um com três argumentos especificados (linha 14) e um com três argumentos inválidos especificados (linha 15). Em seguida, o programa exibe cada objeto nos formatos de data/hora universal e data/hora padrão.

Notas relacionadas com as funções `get` e `set` e o construtor da classe Time

As funções `get` e `set` da classe Time são chamadas por todo o corpo da classe. Em particular, a função `setTime` (linhas 21–26 da Figura 9.9) chama as funções `setHour`, `setMinute` e `setSecond`, e as funções `printUniversal` e `printStandard` chamam as funções `getHour`, `getMinute` e `getSecond` nas linhas 67–68 e 74–76, respectivamente. Em cada caso, essas funções poderiam ter acessado diretamente os dados `private` da classe sem chamar as funções `set` e `get`. Mas considere a possibilidade de alterar a representação da hora de três valores `int` (requerendo 12 bytes de memória) para um único valor `int` a fim de representar o número total de segundos que se passou desde a meia-noite (requerendo quatro bytes de memória). Se fizéssemos essa alteração, somente o corpo das funções que acessam os dados `private` diretamente precisaria mudar — em particular, as funções `set` e `get` individuais para `hour`, `minute` e `second`. Não há nenhuma necessidade de modificar o corpo das funções `setTime`, `printUniversal` ou `printStandard`, porque elas não acessam os dados diretamente. Projetar a classe dessa maneira reduz a probabilidade de erros de programação ao alterar a implementação da classe.

```

1 // Figura 9.10: fig09_10.cpp
2 // Demonstrando um construtor-padrão para a classe Time.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Time.h" // inclui a definição da classe Time a partir de Time.h
8
9 int main()
10 {
11     Time t1; // todos os argumentos convertidos para sua configuração-padrão
12     Time t2( 2 ); // hour especificada; minute e second convertidos para o padrão
13     Time t3( 21, 34 ); // hour e minute especificados; second convertido para o padrão
14     Time t4( 12, 25, 42 ); // hour, minute e second especificados
15     Time t5( 27, 74, 99 ); // valores inválidos especificados
16
17     cout << "Constructed with:\n\nt1: all arguments defaulted\n ";
18     t1.printUniversal(); // 00:00:00
19     cout << "\n ";
20     t1.printStandard(); // 12:00:00 AM
21
22     cout << "\n\nt2: hour specified; minute and second defaulted\n ";
23     t2.printUniversal(); // 02:00:00
24     cout << "\n ";
25     t2.printStandard(); // 2:00:00 AM
26
27     cout << "\n\nt3: hour and minute specified; second defaulted\n ";
28     t3.printUniversal(); // 21:34:00
29     cout << "\n ";
30     t3.printStandard(); // 9:34:00 PM
31
32     cout << "\n\nt4: hour, minute and second specified\n ";
33     t4.printUniversal(); // 12:25:42
34     cout << "\n ";
35     t4.printStandard(); // 12:25:42 PM
36
37     cout << "\n\nt5: all invalid values specified\n ";
38     t5.printUniversal(); // 00:00:00
39     cout << "\n ";
40     t5.printStandard(); // 12:00:00 AM
41     cout << endl;
42     return 0;
43 } // fim de main

```

Constructed with:

t1: all arguments defaulted
00:00:00
12:00:00 AM

t2: hour specified; minute and second defaulted
02:00:00
2:00:00 AM

Figura 9.10 Construtor com argumentos-padrão.

(continua)

```

t3: hour and minute specified; second defaulted
    21:34:00
    9:34:00 PM

t4: hour, minute and second specified
    12:25:42
    12:25:42 PM

t5: all invalid values specified
    00:00:00
    12:00:00 AM

```

Figura 9.10 Construtor com argumentos-padrão.

(continuação)

De maneira semelhante, o construtor de `Time` poderia ser escrito para incluir uma cópia das instruções apropriadas a partir da função `setTime`. Fazer isso talvez seja um pouco mais eficiente, porque a chamada extra ao construtor e a chamada a `setTime` são eliminadas. Entretanto, duplicar instruções em múltiplas funções ou construtores dificulta a alteração da representação interna de dados da classe. Fazer o construtor de `Time` chamar a função `setTime` diretamente requer que qualquer alteração na implementação de `setTime` seja feita somente uma vez.



Erro comum de programação 9.2

Um construtor pode chamar outras funções-membro da classe, como funções `set` e `get`, mas, como ele está inicializando o objeto, os membros de dados ainda podem não estar em um estado consistente. Utilizar os membros de dados antes de eles serem adequadamente inicializados pode causar erros de lógica.

9.7 Destruítores

Um **destrutor** é outro tipo de função-membro especial. O nome do destrutor de uma classe é o **caractere til (~)** seguido pelo nome de classe. Essa convenção de atribuição de nomes é um recurso intuitivo, porque, como veremos mais adiante, o til é o operador de complemento de bit a bit, e, de certo modo, o destrutor é o complemento do construtor. Observe que um destrutor é muitas vezes referido com a abreviação ‘dtor’ na literatura. Preferimos não utilizar essa abreviação.

O destrutor de uma classe é chamado implicitamente quando um objeto é destruído. Isso ocorre, por exemplo, logo que um objeto automático é destruído quando a execução do programa deixa o escopo em que esse objeto foi instanciado. *O destrutor em si não libera a memória do objeto* — ele faz a **faxina de terminação** antes de o sistema reivindicar a memória do objeto, assim a memória pode ser reutilizada para armazenar novos objetos.

Um destrutor não recebe parâmetros nem retorna um valor. Um destrutor pode não especificar um tipo de retorno — nem mesmo `void`. Uma classe pode ter somente um único destrutor — a sobrecarga de destrutor não é permitida.



Erro comum de programação 9.3

É um erro de sintaxe tentar passar argumentos para um destrutor, especificar um tipo de retorno para um destrutor (mesmo `void` não pode ser especificado), retornar valores de um destrutor ou sobrecarregar um destrutor.

Apesar de não fornecermos destrutores para as classes apresentadas até agora, toda classe tem um destrutor. Se o programador não fornecer um destrutor explicitamente, o compilador cria um destrutor ‘vazio’. [Nota: Veremos que este destrutor criado implicitamente, de fato, realiza operações importantes em objetos criados por meio de composição (Capítulo 10) e herança (Capítulo 12). No Capítulo 11, construiremos destrutores apropriados para classes cujos objetos contêm memória alocada dinamicamente (por exemplo, arrays e strings) ou utilizam outros recursos do sistema (por exemplo, arquivos em disco, que estudaremos no Capítulo 17). Discutimos como alocar e desalocar dinamicamente memória no Capítulo 10.



Observação de engenharia de software 9.11

Como veremos no restante do livro, construtores e destrutores têm uma proeminência muito maior em C++ e na programação orientada a objetos do que a ensinada aqui apenas com a nossa breve introdução.

9.8 Quando construtores e destrutores são chamados

Os construtores e destrutores são chamados implicitamente pelo compilador. A ordem em que essas chamadas de função ocorrem depende da ordem em que a execução entra e sai dos escopos em que os objetos são instanciados. Geralmente, as chamadas de destrutor são feitas na ordem inversa das chamadas de construtor correspondentes, mas, como veremos nas figuras 9.11–9.13, as classes de armazenamento de objetos podem alterar a ordem em que destrutores são chamados.

Os construtores são chamados para objetos definidos no escopo global antes de qualquer outra função (incluindo `main`) nesse arquivo começar a execução (embora a ordem de execução de construtores de objeto global entre arquivos não seja garantida). Os destrutores correspondentes serão chamados quando `main` terminar. A função `exit` força um programa a terminar imediatamente e não executa os destrutores de objetos automáticos. A função é frequentemente utilizada para terminar um programa quando um erro é detectado na entrada ou quando não é possível abrir um arquivo a ser processado pelo programa. A função `abort` trabalha de maneira semelhante à função `exit`, mas força o programa a terminar imediatamente, sem permitir que os destrutores de quaisquer objetos sejam chamados. A função `abort` é normalmente utilizada para indicar uma terminação anormal do programa. (Ver o Capítulo 24, “Outros tópicos”, para obter informações adicionais sobre as funções `exit` e `abort`.)

O construtor de um objeto local automático é chamado quando a execução alcança o ponto em que o objeto é definido — o destrutor correspondente é chamado quando a execução deixa o escopo do objeto (isto é, quando o bloco em que esse objeto é definido tiver terminado a execução). Os construtores e destrutores para objetos automáticos são chamados toda vez que a execução entra e sai do escopo do objeto. Os destrutores não serão chamados para objetos automáticos se o programa terminar com uma chamada à função `exit` ou à função `abort`.

O construtor de um objeto local `static` é chamado uma única vez, logo que a execução alcança o ponto em que o objeto foi definido — o destrutor correspondente é chamado quando `main` termina ou quando o programa chama a função `exit`. Os objetos globais e `static` são destruídos na ordem inversa de sua criação. Os destrutores não são chamados para os objetos `static` se o programa terminar com uma chamada à função `abort`.

O programa das figuras 9.11–9.13 demonstra a ordem em que os construtores e destrutores são indicados para os objetos da classe `CreateAndDestroy` (figuras 9.11 e 9.12) de várias classes de armazenamento em diversos escopos. Cada objeto da classe `CreateAndDestroy` contém (linhas 16–17) um inteiro (`objectId`) e uma `string` (`message`) que são utilizados na saída do programa para identificar o objeto. Esse exemplo mecânico é puramente para propósitos pedagógicos. Por essa razão, a linha 23 do destrutor na Figura 9.12 determina se o objeto que está sendo destruído tem um valor `objectId` 1 ou 6 e, se tiver, gera saída de um caractere de nova linha. Essa linha ajuda a tornar a saída do programa mais fácil de seguir.

A Figura 9.13 define o objeto `first` (linha 12) no escopo global. Seu construtor na realidade é chamado antes que qualquer instrução em `main` seja executada e seu destrutor é chamado na terminação do programa depois de os destrutores de todos os outros objetos executarem.

A função `main` (linhas 14–26) declara três objetos. Os objetos `second` (linha 17) e `fourth` (linha 23) são objetos automáticos locais e o objeto `third` (linha 18) é um objeto local `static`. O construtor para cada um desses objetos é chamado quando a execução alcança o ponto em que esse objeto é declarado. Os destrutores para objetos `fourth` e, então, `second` são chamados (isto é, o inverso da ordem em que seus construtores foram chamados) quando a execução alcança o fim de `main`. Como o objeto `third` é `static`, ele existe até a terminação do programa. O destrutor do objeto `third` é chamado antes do destrutor para o objeto global `first`, mas depois de todos os outros objetos serem destruídos.

```

1 // Figura 9.11: CreateAndDestroy.h
2 // Definição da classe CreateAndDestroy.
3 // Funções-membro definidas em CreateAndDestroy.cpp.
4 #include <string>
5 using std::string;
6
7 #ifndef CREATE_H
8 #define CREATE_H
9
10 class CreateAndDestroy
11 {
12 public:
13     CreateAndDestroy( int, string ); // construtor
14     ~CreateAndDestroy(); // destrutor
15 private:
16     int objectId; // Número de ID do objeto

```

Figura 9.11 Definição da classe `CreateAndDestroy`.

(continua)

```

17     string message; // mensagem descrevendo o objeto
18 }; // fim da classe CreateAndDestroy
19
20 #endif

```

Figura 9.11 Definição da classe CreateAndDestroy.

(continuação)

```

1 // Figura 9.12: CreateAndDestroy.cpp
2 // Definições de função-membro da classe CreateAndDestroy.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "CreateAndDestroy.h" // inclui a definição da classe CreateAndDestroy
8
9 // construtor
10 CreateAndDestroy::CreateAndDestroy( int ID, string messageString )
11 {
12     objectID = ID; // configura o número de ID do objeto
13     message = messageString; // configura mensagem descritiva do objeto
14
15     cout << "Object " << objectID << "   constructor runs   "
16         << message << endl;
17 } // fim do construtor CreateAndDestroy
18
19 // destrutor
20 CreateAndDestroy::~CreateAndDestroy()
21 {
22     // gera saída de nova linha para certos objetos; ajuda a legibilidade
23     cout << ( objectID == 1 || objectID == 6 ? "\n" : "" );
24
25     cout << "Object " << objectID << "   destructor runs   "
26         << message << endl;
27 } // fim do destrutor ~CreateAndDestroy

```

Figura 9.12 Definições de função-membro da classe CreateAndDestroy.

```

1 // Figura 9.13: fig09_13.cpp
2 // Demonstrando a ordem em que construtores e
3 // destrutores são chamados.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include "CreateAndDestroy.h" // inclui a definição da classe CreateAndDestroy
9
10 void create( void ); // protótipo
11
12 CreateAndDestroy first( 1, "(global before main)" ); // objeto global
13
14 int main()
15 {

```

Figura 9.13 A ordem em que construtores e destrutores são chamados.

(continua)


```

16     cout << "\nMAIN FUNCTION: EXECUTION BEGINS" << endl;
17     CreateAndDestroy second( 2, "(local automatic in main)" );
18     static CreateAndDestroy third( 3, "(local static in main)" );
19
20     create(); // chama função para criar objetos
21
22     cout << "\nMAIN FUNCTION: EXECUTION RESUMES" << endl;
23     CreateAndDestroy fourth( 4, "(local automatic in main)" );
24     cout << "\nMAIN FUNCTION: EXECUTION ENDS" << endl;
25     return 0;
26 } // fim de main
27
28 // função para criar objetos
29 void create( void )
30 {
31     cout << "\nCREATE FUNCTION: EXECUTION BEGINS" << endl;
32     CreateAndDestroy fifth( 5, "(local automatic in create)" );
33     static CreateAndDestroy sixth( 6, "(local static in create)" );
34     CreateAndDestroy seventh( 7, "(local automatic in create)" );
35     cout << "\nCREATE FUNCTION: EXECUTION ENDS" << endl;
36 } // fim da função create

```

Object 1 constructor runs (global before main)

MAIN FUNCTION: EXECUTION BEGINS

Object 2 constructor runs (local automatic in main)

Object 3 constructor runs (local static in main)

CREATE FUNCTION: EXECUTION BEGINS

Object 5 constructor runs (local automatic in create)

Object 6 constructor runs (local static in create)

Object 7 constructor runs (local automatic in create)

CREATE FUNCTION: EXECUTION ENDS

Object 7 destructor runs (local automatic in create)

Object 5 destructor runs (local automatic in create)

MAIN FUNCTION: EXECUTION RESUMES

Object 4 constructor runs (local automatic in main)

MAIN FUNCTION: EXECUTION ENDS

Object 4 destructor runs (local automatic in main)

Object 2 destructor runs (local automatic in main)

Object 6 destructor runs (local static in create)

Object 3 destructor runs (local static in main)

Object 1 destructor runs (global before main)

Figura 9.13 A ordem em que construtores e destrutores são chamados.

(continuação)

A função `create` (linhas 29–36) declara três objetos — `fifth` (linha 32) e `seventh` (linha 34) como objetos automáticos locais e `sixth` (linha 33) como um objeto local `static`. Os destrutores dos objetos `seventh` e, em seguida, `fifth` são chamados (isto é, a ordem inversa em que seus construtores foram chamados) quando `create` termina. Como `sixth` é `static`, ele existe até a terminação do programa. O destrutor de `sixth` é chamado antes dos destrutores de `third` e `first`, mas depois de todos os outros objetos serem destruídos.

9.9 Estudo de caso da classe Time: uma armadilha sutil — retornar uma referência a um membro de dados `private`

Uma referência a um objeto é um alias para o nome do objeto e, portanto, pode ser utilizado à esquerda de uma instrução de atribuição. Nesse contexto, a referência faz um *lvalue* perfeitamente aceitável que pode receber um valor. Uma maneira de utilizar essa capacidade (infelizmente!) é fazer uma função-membro `public` de uma classe retornar uma referência a um membro de dados `private` dessa classe. Observe que, se uma função retornar uma referência `const`, esta não pode ser utilizada como um *lvalue* modificável.

O programa das figuras 9.14–9.16 utiliza uma classe `Time` simplificada (figuras 9.14 e 9.15) para demonstrar o retorno de uma referência a um membro de dados `private` com a função-membro `badSetHour` (declarada na Figura 9.14 na linha 15 e definida na Figura 9.15 nas linhas 29–33). Esse retorno de referência na realidade torna uma chamada à função-membro `badSetHour` um alias para o membro de dados `private` `hour`! A chamada de função pode ser utilizada de todas as maneiras que o membro de dados `private` pode ser utilizado, inclusive como um *lvalue* em uma instrução de atribuição, permitindo, assim, que os clientes da classe sobrescrevam acidentalmente os dados `private` da classe à vontade! Observe que o mesmo problema ocorreria se um ponteiro para os dados `private` precisasse ser retornado pela função.

A Figura 9.16 declara o objeto `Time t` (linha 12) e a referência `hourRef` (linha 15), que é inicializada com a referência retornada pela chamada `t.badSetHour(20)`. A linha 17 exibe o valor do alias `hourRef`. Isso mostra como `hourRef` quebra o encapsulamento da classe — as instruções em `main` não devem ter acesso aos dados `private` da classe. Em seguida, a linha 18 utiliza o alias para configurar o valor de `hour` como 30 (um valor inválido) e a linha 19 exibe o valor retornado pela função `getHour` para mostrar que atribuir um valor a `hourRef` realmente modifica os dados `private` no objeto `Time t`. Por fim, a linha 23 utiliza a própria chamada de função `badSetHour` como um *lvalue* e atribui 74 (outro valor inválido) à referência retornada pela função. A linha 28 exibe novamente o valor retornado pela função `getHour` para mostrar que atribuir um valor ao resultado da chamada de função na linha 23 modifica os dados `private` no objeto `Time t`.

```

1 // Figura 9.14: Time.h
2 // Declaração da classe Time.
3 // Funções-membro definidas em Time.cpp
4
5 // impede múltiplas inclusões de arquivo de cabeçalho
6 #ifndef TIME_H
7 #define TIME_H
8
9 class Time
10 {
11 public:
12     Time( int = 0, int = 0, int = 0 );
13     void setTime( int, int, int );
14     int getHour();
15     int &badSetHour( int ); // retorno de referência PERIGOSO
16 private:
17     int hour;
18     int minute;
19     int second;
20 }; // fim da classe Time
21
22 #endif

```

Figura 9.14 Retornando uma referência a um membro de dados `private`.

```

1 // Figura 9.15: Time.cpp
2 // Definições de função-membro para a classe Time.
3 #include "Time.h" // inclui definição da classe Time
4
5 // função constructor para inicializar dados private;

```

Figura 9.15 Retornando uma referência a um membro de dados `private`.

(continua)

```

6 // chama a função-membro setTime para configurar variáveis;
7 // valores-padrão são 0 (ver definição de classe)
8 Time::Time( int hr, int min, int sec )
9 {
10     setTime( hr, min, sec );
11 } // fim do construtor de Time
12
13 // configura valores de hour, minute e second
14 void Time::setTime( int h, int m, int s )
15 {
16     hour = ( h >= 0 && h < 24 ) ? h : 0; // valida horas
17     minute = ( m >= 0 && m < 60 ) ? m : 0; // valida minutos
18     second = ( s >= 0 && s < 60 ) ? s : 0; // valida segundos
19 } // fim da função setTime
20
21 // retorna valor de hour
22 int Time::getHour()
23 {
24     return hour;
25 } // fim da função getHour
26
27 // PRÁTICA DE PROGRAMAÇÃO RUIM:
28 // Retornar uma referência a um membro de dados private.
29 int &Time::badSetHour( int hh )
30 {
31     hour = ( hh >= 0 && hh < 24 ) ? hh : 0;
32     return hour; // retorno de referência PERIGOSO
33 } // fim da função badSetHour

```

Figura 9.15 Retornando uma referência a um membro de dados private.

(continuação)

```

1 // Figura 9.16: fig09_16.cpp
2 // Demonstrando uma função-membro public que
3 // retorna uma referência a um membro de dados private.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include "Time.h" // inclui definição da classe Time
9
10 int main()
11 {
12     Time t; // cria o objeto Time
13
14     // inicializa hourRef com a referência retornada por badSetHour
15     int &hourRef = t.badSetHour( 20 ); // 20 is a valid hour
16
17     cout << "Valid hour before modification: " << hourRef;
18     hourRef = 30; // use hourRef to set invalid value in Time object t
19     cout << "\nInvalid hour after modification: " << t.getHour();
20
21     // Dangerous: Function call that returns
22     // a reference can be used as an lvalue!
23     t.badSetHour( 12 ) = 74; // assign another invalid value to hour

```

Figura 9.16 Retornando uma referência a um membro de dados private.

(continua)

```

24
25     cout << "\n\n*****\n"
26         << "POOR PROGRAMMING PRACTICE!!!!!!\n"
27         << "t.badSetHour( 12 ) as an lvalue, invalid hour: "
28         << t.getHour()
29         << "\n*****" << endl;
30     return 0;
31 } // fim de main

```

Valid hour before modification: 20
Invalid hour after modification: 30

```

*****
POOR PROGRAMMING PRACTICE!!!!!!
t.badSetHour( 12 ) as an lvalue, invalid hour: 74
*****

```

Figura 9.16 Retornando uma referência a um membro de dados private.

(continuação)



Dica de prevenção de erro 9.4

Retornar uma referência ou um ponteiro para um membro de dados private quebra o encapsulamento da classe e torna o código-cliente dependente da representação dos dados da classe. Portanto, retornar ponteiros ou referências aos dados private é uma prática perigosa que deve ser evitada.

9.10 Atribuição-padrão de membro a membro

O operador de atribuição (=) pode ser utilizado para atribuir um objeto a outro objeto do mesmo tipo. Por padrão, tal atribuição é realizada pela **atribuição de membro a membro** — cada membro de dados do objeto à direita do operador de atribuição é atribuído individualmente ao mesmo membro de dados no objeto à esquerda do operador de atribuição. As figuras 9.17–9.18 definem a classe Date para ser utilizada neste exemplo. A linha 20 da Figura 9.19 utiliza a atribuição-padrão de membro a membro para atribuir os membros de dados

```

1 // Figura 9.17: Date.h
2 // Declaração da classe Date.
3 // Funções-membro são definidas em Date.cpp
4
5 // impede múltiplas inclusões de arquivo de cabeçalho
6 #ifndef DATE_H
7 #define DATE_H
8
9 // definição da classe Date
10 class Date
11 {
12 public:
13     Date( int = 1, int = 1, int = 2000 ); // construtor-padrão
14     void print();
15 private:
16     int month;
17     int day;
18     int year;
19 }; // fim da classe Date
20
21 #endif

```

Figura 9.17 Arquivo de cabeçalho da classe Date.

```

1 // Figura 9.18: Date.cpp
2 // Definições de função-membro da classe Date.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Date.h" // inclui a definição da classe Date a partir de Date.h
8
9 // construtor Date (deve fazer verificação de intervalo)
10 Date::Date( int m, int d, int y )
11 {
12     month = m;
13     day = d;
14     year = y;
15 } // fim do construtor Date
16
17 // imprime Date no formato mm/dd/aaaa
18 void Date::print()
19 {
20     cout << month << '/' << day << '/' << year;
21 } // fim da função print

```

Figura 9.18 Definições de função-membro da classe Date.

do objeto Date `date1` aos membros de dados correspondentes do objeto Date `date2`. Nesse caso, o membro `month` de `date1` é atribuído ao membro `month` de `date2`, o membro `day` de `date1` é atribuído ao membro `day` de `date2` e o membro `year` de `date1` é atribuído ao membro `year` de `date2`. *[Atenção: A atribuição de membro a membro pode causar problemas sérios quando utilizada com uma classe cujos membros de dados contêm ponteiros para a memória alocada dinamicamente; discutimos esses problemas no Capítulo 11 e mostramos como lidar com eles.]* Note que o construtor Date não contém nenhuma verificação de erros; deixamos isso para os exercícios.

Os objetos podem ser passados como funções argumentos e podem ser retornados a partir de funções. Essa passagem e esse retorno são realizados utilizando a passagem por valor por padrão — uma cópia do objeto é passada ou retornada. Nesses casos, o C++ cria um novo objeto e utiliza um **construtor de cópia** para copiar os valores do objeto original para o novo objeto. Para cada classe, o compilador fornece um construtor de cópia-padrão que copia cada membro do objeto original para o membro correspondente do novo objeto. Assim como a atribuição de membro a membro, os construtores de cópia podem causar sérios problemas quando utilizados com uma classe cujos membros de dados contêm ponteiros para memória dinamicamente alocada. O Capítulo 11 discute como programadores podem definir um construtor de cópia personalizado que copia de maneira adequada os objetos contendo ponteiros para a memória dinamicamente alocada.

```

1 // Figura 9.19: fig09_19.cpp
2 // Demonstrando que os objetos de classe podem ser atribuídos
3 // um ao outro utilizando atribuição-padrão de membro a membro.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include "Date.h" // inclui a definição da classe Date a partir de Date.h
9
10 int main()
11 {
12     Date date1( 7, 4, 2004 );
13     Date date2; // date2 assume padrão de 1/1/2000
14
15     cout << "date1 = ";

```

Figura 9.19 Atribuição-padrão de membro a membro.

(continua)

```

16     date1.print();
17     cout << "\ndate2 = ";
18     date2.print();
19
20     date2 = date1; // atribuição-padrão de membro a membro
21
22     cout << "\n\nAfter default memberwise assignment, date2 = ";
23     date2.print();
24     cout << endl;
25     return 0;
26 } // fim de main

```

```

date1 = 7/4/2004
date2 = 1/1/2000

```

```

After default memberwise assignment, date2 = 7/4/2004

```

Figura 9.19 Atribuição-padrão de membro a membro.

(continuação)



Dica de desempenho 9.3

A passagem de um objeto por valor é boa de um ponto de vista de segurança, porque a função chamada não tem acesso ao objeto original no chamador, mas pode degradar o desempenho ao fazer uma cópia de um objeto grande. Um objeto pode ser passado por referência passando um ponteiro ou uma referência para o objeto. A passagem por referência oferece bom desempenho, mas é mais fraca de um ponto de vista de segurança, porque a função chamada recebe acesso ao objeto original. A passagem por referência `const` é uma alternativa segura de bom desempenho (essa pode ser implementada com um parâmetro de referência `const` ou com um parâmetro de ponteiro para dados `const`).

9.11 Reusabilidade de software

As pessoas que escrevem programas orientados a objetos se concentram na implementação de classes úteis. Há uma enorme motivação em capturar e catalogar classes para que possam ser acessadas por grandes segmentos da comunidade de programação. Existem muitas bibliotecas de classe substanciais e outras estão sendo desenvolvidas em todo o mundo. Os softwares estão sendo construídos, cada vez mais, a partir de componentes existentes, bem definidos, cuidadosamente testados, bem documentados, portáteis, de alto desempenho e amplamente disponíveis. Esse tipo de reusabilidade de software acelera o desenvolvimento de software poderoso e de alta qualidade. O **desenvolvimento rápido de aplicativos** (*rapid applications development – RAD*) por meio dos mecanismos de componentes reutilizáveis tornou-se um campo importante.

Entretanto, problemas significativos devem ser resolvidos antes de o potencial de reusabilidade de software ser realizado. Precisamos de esquemas de catalogação e licenciamento, mecanismos de proteção para assegurar que cópias-mestre de classes não estejam corrompidas, esquemas de descrição para que designers de novos sistemas possam determinar facilmente se objetos existentes atendem às suas necessidades, mecanismos de navegação para determinar quais classes estão disponíveis e o rigor com que atendem aos requisitos do desenvolvedor de software etc. Muitos problemas interessantes de pesquisa e desenvolvimento precisam ser resolvidos. Há grande motivação em resolver esses problemas, porque o valor potencial de suas soluções é enorme.

9.12 Estudo de caso de engenharia de software: começando a programar as classes do sistema ATM (opcional)

Nas seções “Estudo de caso de engenharia de software” dos capítulos 1 a 7, introduzimos os fundamentos da orientação a objetos e desenvolvemos um projeto orientado a objetos para nosso sistema ATM. Anteriormente neste capítulo, discutimos vários detalhes sobre a programação com classes C++. Agora, iniciaremos a implementação do nosso projeto orientado a objetos em C++. No final desta seção, mostramos como converter diagramas de classes em arquivos de cabeçalho C++. No final da seção “Estudo de caso de engenharia de software” (Seção 13.10), modificaremos os arquivos de cabeçalho para incorporar o conceito orientado a objetos de herança. Apresentamos a completa implementação de código C++ no Apêndice G.

Visibilidade

Agora, aplicamos especificadores de acesso aos membros das nossas classes. No Capítulo 3, introduzimos os especificadores de acesso `public` e `private`. Os modificadores de acesso determinam a **visibilidade** ou acessibilidade dos atributos e operações de um objeto a

outros objetos. Antes de iniciarmos a implementação do nosso projeto, devemos considerar quais atributos e operações das nossas classes devem ser `public` e quais devem ser `private`.

No Capítulo 3, observamos que membros de dados devem ser normalmente `private` e que as funções-membro invocadas por clientes de uma dada classe devem ser `public`. Entretanto, as funções-membro chamadas apenas por outras funções-membro da classe como ‘funções utilitárias’ devem ser normalmente `private`. A UML emprega **marcadores de visibilidade** para modelar a visibilidade dos atributos e operações. Visibilidade pública é indicada colocando-se um sinal de adição (+) antes de uma operação ou atributo; um sinal de subtração (–) indica visibilidade privada. A Figura 9.20 mostra nosso diagrama de classes atualizado com marcadores de visibilidade incluídos. [Nota: Não incluímos parâmetros de operação na Figura 9.20. Isso é perfeitamente normal. Adicionar marcadores de visibilidade não afeta os parâmetros já modelados nos diagramas de classes das figuras 6.22–6.25.]

Navegabilidade

Antes de começarmos a implementar nosso projeto em C++, apresentaremos uma notação da UML adicional. O diagrama de classes na Figura 9.21 refina ainda mais os relacionamentos entre as classes no sistema ATM adicionando setas de navegabilidade às linhas de associação. **Setas de navegabilidade** (representadas como setas com pontas no diagrama de classe) indicam a direção em que uma associação pode ser percorrida e são baseadas nas colaborações modeladas em diagramas de comunicação e de seqüências (ver Seção 7.12). Ao implementar um sistema projetado utilizando a UML, os programadores utilizam setas de navegabilidade para ajudar a determinar os objetos que precisam de referências ou ponteiros para outros objetos. Por exemplo, a seta de navegabilidade que aponta da classe ATM para a classe BankDatabase indica que podemos navegar da primeira à última, permitindo assim que a classe ATM invoque as operações da classe BankDatabase. Entretanto, como a Figura 9.21 não contém uma seta de navegabilidade que aponta da classe BankDatabase para a classe ATM, BankDatabase não pode acessar as operações de ATM. Observe que as associações em um diagrama de classes que têm setas de navegabilidade nas duas extremidades ou que simplesmente não têm setas de navegabilidade indicam **navegabilidade bidirecional** — a navegação pode acontecer em qualquer direção pela associação.

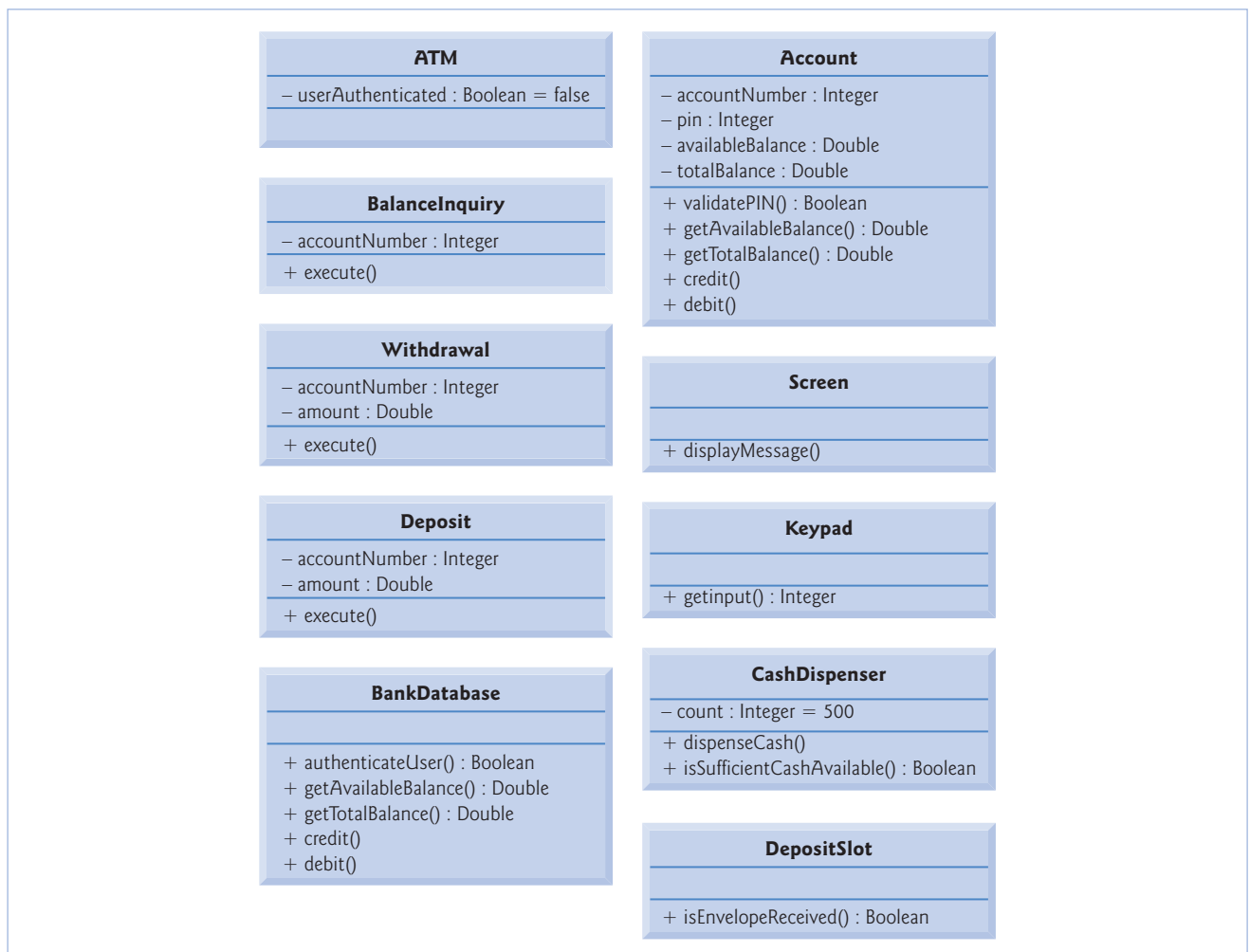


Figura 9.20 Diagrama de classes com marcadores de visibilidade.

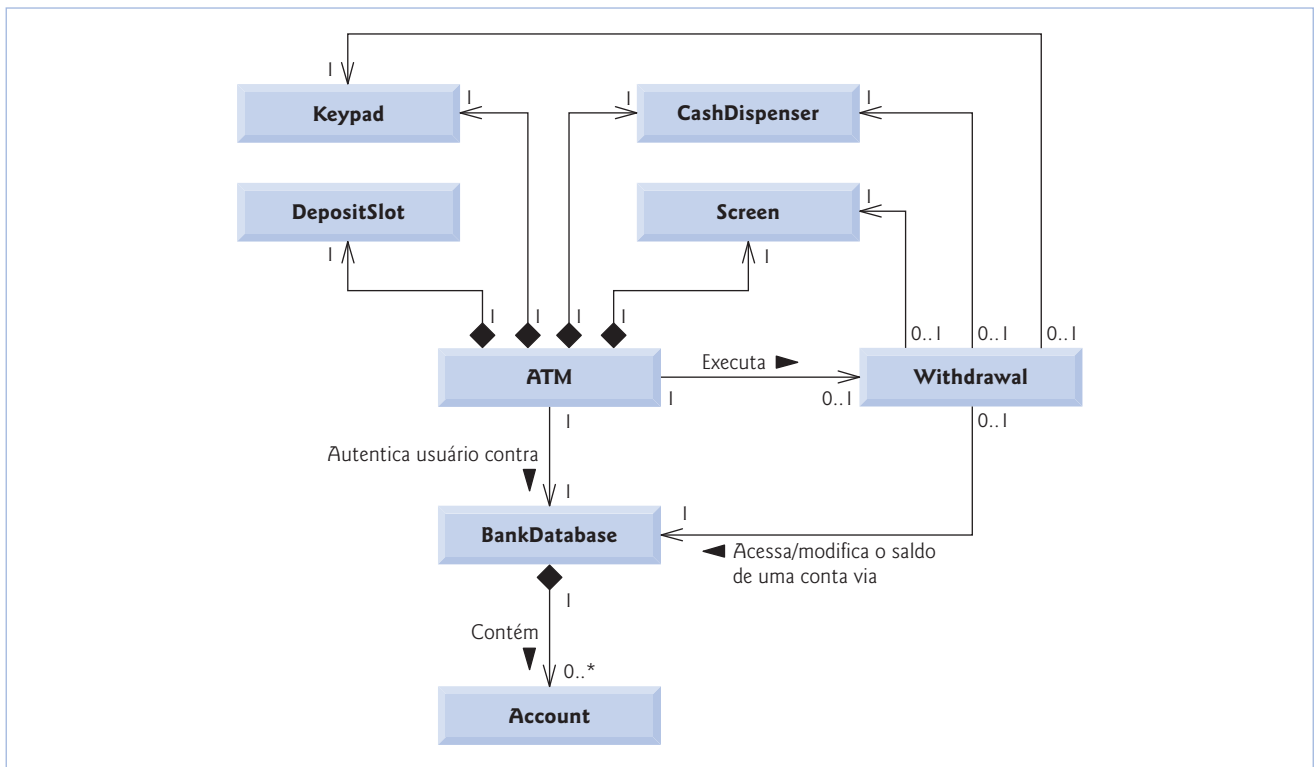


Figura 9.21 Diagrama de classes com setas de navegabilidade.

Como ocorre com o diagrama de classes da Figura 3.23, o diagrama de classes da Figura 9.21 omite as classes *BalanceInquiry* e *Deposit* para manter o diagrama simples. A navegabilidade das associações de que essas classes participam é bem parecida com a navegabilidade das associações da classe *Withdrawal*. Lembre-se, a partir do que foi discutido na Seção 3.11, de que *BalanceInquiry* tem uma associação com a classe *Screen*. Podemos navegar da classe *BalanceInquiry* para a classe *Screen* ao longo dessa associação, mas não podemos navegar da classe *Screen* para a classe *BalanceInquiry*. Portanto, se fôssemos modelar a classe *BalanceInquiry* na Figura 9.21, colocaríamos uma seta de navegabilidade no fim da classe *Screen* dessa associação. Lembre-se também de que a classe *Deposit* se associa às classes *Screen*, *Keypad* e *DepositSlot*. Podemos navegar da classe *Deposit* para cada uma dessas classes, mas não vice-versa. Portanto, colocaríamos setas de navegabilidade nas extremidades *Screen*, *Keypad* e *DepositSlot* dessas associações. [Nota: Modelamos essas classes e associações adicionais no nosso diagrama de classes final na Seção 13.10, depois de simplificarmos a estrutura do nosso sistema incorporando o conceito de herança ao projeto orientado a objetos.]

Implementando o sistema ATM a partir de seu projeto em UML

Agora, estamos prontos para começar a implementar o sistema ATM. Primeiro convertemos as classes nos diagramas das figuras 9.20 e 9.21 em arquivos de cabeçalho C++. Esse código representará o ‘esqueleto’ do sistema. No Capítulo 13, modificamos os arquivos de cabeçalho para incorporar o conceito de herança orientado a objetos. No Apêndice G, “Código para o estudo de caso do ATM”, apresentamos o código C++ funcional completo para nosso modelo.

Como um exemplo, começamos a desenvolver o arquivo de cabeçalho para a classe *Withdrawal* de nosso design da classe *Withdrawal* da Figura 9.20. Utilizamos essa figura para determinar os atributos e operações da classe. Utilizamos o modelo da UML na Figura 9.21 para determinar as associações entre as classes. Seguimos as cinco diretrizes a seguir para cada classe:

1. Utilize o nome localizado no primeiro compartimento de uma classe em um diagrama de classes para definir a classe em um arquivo de cabeçalho (Figura 9.22). Utilize as diretivas de pré-processador `#ifndef`, `#define` e `#endif` para impedir que o arquivo de cabeçalho seja incluído mais de uma vez em um programa.
2. Utilize os atributos localizados no segundo compartimento da classe para declarar os membros de dados. Por exemplo, os atributos `private accountNumber` e `amount` da classe *Withdrawal* produzem o código na Figura 9.23.
3. Utilize as associações descritas no diagrama de classes para declarar referências (ou ponteiros, onde apropriado) a outros objetos. Por exemplo, de acordo com a Figura 9.21, *Withdrawal* pode acessar um objeto da classe *Screen*, um objeto da classe *Keypad*, um objeto da classe *CashDispenser* e um objeto de classe *BankDatabase*. A classe *Withdrawal* deve manter handles nesses objetos para enviar-lhes mensagens, portanto, as linhas 19–22 da Figura 9.24 declaram quatro referências como membros de dados `private`. Na implementação da classe *Withdrawal* no Apêndice G, um construtor inicializa esses membros de dados

```

1 // Figura 9.22: Withdrawal.h
2 // Definição da classe Withdrawal que representa uma transação de retirada
3 #ifndef WITHDRAWAL_H
4 #define WITHDRAWAL_H
5
6 class Withdrawal
7 {
8 }; // fim da classe Withdrawal
9
10 #endif // WITHDRAWAL_H

```

Figura 9.22 Definição da classe `Withdrawal` incluída em empacotadores de pré-processador.

```

1 // Figura 9.23: Withdrawal.h
2 // Definição da classe Withdrawal que representa uma transação de retirada
3 #ifndef WITHDRAWAL_H
4 #define WITHDRAWAL_H
5
6 class Withdrawal
7 {
8 private:
9     // atributos
10    int accountNumber; // conta a sacar fundos
11    double amount; // quantia a sacar
12 }; // fim da classe Withdrawal
13
14 #endif // WITHDRAWAL_H

```

Figura 9.23 Adicionando atributos ao arquivo de cabeçalho de classe `Withdrawal`.

com referência a objetos reais. Observe que as linhas 6–9 incluem (`#include`) os arquivos de cabeçalho contendo as definições das classes `Screen`, `Keypad`, `CashDispenser` e `BankDatabase` para podermos declarar as referências aos objetos dessas classes nas linhas 19–22.

- Acabamos descobrindo que incluir os arquivos de cabeçalho às classes `Screen`, `Keypad`, `CashDispenser` e `BankDatabase` na Figura 9.24 faz mais do que é necessário. A classe `Withdrawal` contém *referências* a objetos dessas classes — ela não contém objetos reais — e a quantidade de informações requeridas pelo compilador para criar uma referência difere daquela que é requerida para criar um objeto. Lembre-se de que criar um objeto requer fornecer ao compilador uma definição da classe que introduza o nome da classe como um novo tipo definido pelo usuário e que indique os membros de dados que determinam a quantidade de memória necessária para armazenar o objeto. Declarar uma *referência* (ou ponteiro) a um objeto, porém, requer somente que o compilador saiba que a classe do objeto existe — ele não precisa saber o tamanho do objeto. Qualquer referência (ou ponteiro), independentemente da classe do objeto à qual ela se refere, contém apenas o endereço de memória do objeto real. A quantidade de memória requerida para o armazenamento de um endereço é uma característica física do hardware do computador. Assim o compilador sabe o tamanho de qualquer referência (ou ponteiro). Como resultado, é desnecessário incluir o arquivo de cabeçalho completo de uma classe ao declarar apenas uma referência a um objeto dessa classe — precisamos introduzir o nome da classe, mas não precisamos fornecer o layout de dados do objeto, porque o compilador já sabe o tamanho de todas as referências. O C++ fornece uma instrução chamada **declaração antecipada** [*forward declaration*] que significa que um arquivo de cabeçalho contém referências ou ponteiros para uma classe, mas que a definição de classe reside fora do arquivo de cabeçalho. Podemos substituir os `#includes` na definição da classe `Withdrawal` da Figura 9.24 pelas declarações antecipadas das classes `Screen`, `Keypad`, `CashDispenser` e `BankDatabase` (linhas 6–9 na Figura 9.25). Em vez de incluir (`#include`) o arquivo de cabeçalho inteiro em cada uma dessas classes, colocamos apenas uma declaração antecipada de cada classe no arquivo de cabeçalho da classe `Withdrawal`. Observe que, se a classe `Withdrawal` contivesse objetos reais em vez de referências (isto é, se os sinais de `&` nas linhas 19–22 fossem omitidos), então, de fato, precisaríamos incluir (`#include`) os arquivos de cabeçalho completos.

```

1 // Figura 9.24: Withdrawal.h
2 // Definição da classe Withdrawal que representa uma transação de retirada
3 #ifndef WITHDRAWAL_H
4 #define WITHDRAWAL_H
5
6 #include "Screen.h" // inclui a definição da classe Screen
7 #include "Keypad.h" // inclui a definição da classe Keypad
8 #include "CashDispenser.h" // inclui a definição da classe CashDispenser
9 #include "BankDatabase.h" // inclui a definição da classe BankDatabase
10
11 class Withdrawal
12 {
13 private:
14     // atributos
15     int accountNumber; // conta a sacar fundos
16     double amount; // quantia a sacar
17
18     // referências a objetos associados
19     Screen &screen; // referência à tela do ATM
20     Keypad &keypad; // referência ao teclado do ATM
21     CashDispenser &cashDispenser; // referência ao dispensador de notas do ATM
22     BankDatabase &bankDatabase; // referência ao banco de dados de info de conta
23 }; // fim da classe Withdrawal
24
25 #endif // WITHDRAWAL_H

```

Figura 9.24 Declarando referências aos objetos associados com a classe `Withdrawal`.

```

1 // Figura 9.25: Withdrawal.h
2 // Definição da classe Withdrawal que representa uma transação de retirada
3 #ifndef WITHDRAWAL_H
4 #define WITHDRAWAL_H
5
6 class Screen; // declaração antecipada da classe Screen
7 class Keypad; // declaração antecipada da classe Keypad
8 class CashDispenser; // declaração antecipada da classe CashDispenser
9 class BankDatabase; // declaração antecipada da classe BankDatabase
10
11 class Withdrawal
12 {
13 private:
14     // atributos
15     int accountNumber; // conta a sacar fundos
16     double amount; // quantia a sacar
17
18     // referências a objetos associados
19     Screen &screen; // referência à tela do ATM
20     Keypad &keypad; // referência ao teclado do ATM
21     CashDispenser &cashDispenser; // referência ao dispensador de notas do ATM
22     BankDatabase &bankDatabase; // referência ao banco de dados de informações de conta
23 }; // fim da classe Withdrawal
24
25 #endif // WITHDRAWAL_H

```

Figura 9.25 Utilizando declarações antecipadas em vez de diretivas `#include`.

Observe que utilizar uma declaração antecipada (onde possível) em vez de incluir um arquivo de cabeçalho inteiro ajuda a evitar um problema de pré-processador chamado **inclusão circular**. Esse problema ocorre quando o arquivo de cabeçalho de uma classe A inclui (`#includes`) o arquivo de cabeçalho em uma classe B e vice-versa. Alguns pré-processadores não são capazes de resolver essas diretivas `#include`, causando um erro de compilação. Se a classe A, por exemplo, utiliza somente uma referência a um objeto da classe B, então o `#include` no arquivo de cabeçalho da classe A pode ser substituído por uma declaração antecipada da classe B para impedir a inclusão circular.

5. Utilize as operações localizadas no terceiro compartimento da Figura 9.20 para escrever os protótipos de função das funções-membro da classe. Se ainda não especificamos um tipo de retorno para uma operação, declaramos a função-membro com um tipo de retorno `void`. Consulte os diagramas de classes das figuras 6.22–6.25 para declarar qualquer parâmetro necessário. Por exemplo, adicionar a operação `public execute()` na classe `Withdrawal`, que tem uma lista de parâmetros vazia, resulta no protótipo da linha 15 da Figura 9.26. [Nota: Codificamos as definições de funções-membro em arquivos `.cpp` quando implementamos o sistema ATM completo no Apêndice G.]



Observação de engenharia de software 9.12

Várias ferramentas de modelagem UML podem converter projetos baseados em UML em código C++, aumentando consideravelmente a velocidade do processo de implementação. Para obter informações adicionais sobre esses geradores de códigos ‘automáticos’, consulte os recursos na Internet e na Web listados no final da Seção 2.8.

Isso conclui nossa discussão sobre os princípios básicos da geração de arquivos de cabeçalho de classe a partir de diagramas UML. Na seção “Estudo de caso de engenharia de software” final (Seção 3.11), demonstramos como modificar os arquivos de cabeçalho para incorporar o conceito de herança orientado a objetos.

Exercícios de revisão do estudo de caso de engenharia de software

- 9.1 Determine se a seguinte sentença é verdadeira ou falsa e, se falsa, explique por quê: Se um atributo de uma classe estiver marcado com um sinal de subtração (–) em um diagrama de classes, o atributo não estará diretamente acessível fora da classe.

```

1 // Figura 9.26: Withdrawal.h
2 // Definição da classe Withdrawal que representa uma transação de retirada
3 #ifndef WITHDRAWAL_H
4 #define WITHDRAWAL_H
5
6 class Screen; // declaração antecipada da classe Screen
7 class Keypad; // declaração antecipada da classe Keypad
8 class CashDispenser; // declaração antecipada da classe CashDispenser
9 class BankDatabase; // declaração antecipada da classe BankDatabase
10
11 class Withdrawal
12 {
13 public:
14     // operações
15     void execute(); // realiza a transação
16 private:
17     // atributos
18     int accountNumber; // conta a sacar fundos
19     double amount; // quantia a sacar
20
21     // referências a objetos associados
22     Screen &screen; // referência à tela do ATM
23     Keypad &keypad; // referência ao teclado do ATM
24     CashDispenser &cashDispenser; // referência ao dispensador de notas do ATM
25     BankDatabase &bankDatabase; // referência ao banco de dados de informações de conta
26 }; // fim da classe Withdrawal
27
28 #endif // WITHDRAWAL_H

```

Figura 9.26 Adicionando operações ao arquivo de cabeçalho da classe `Withdrawal`.

- 9.2** Na Figura 9.21, a associação entre a ATM e a Screen indica que:
- podemos navegar da Screen para a ATM
 - podemos navegar da ATM para a Screen
 - Tanto a como b; a associação é bidirecional
 - Nenhuma acima
- 9.3** Escreva um código C++ para começar a implementar o projeto da classe Account.

Respostas aos exercícios de revisão do estudo de caso de engenharia de software

- 9.1** Verdadeira. O sinal de subtração (−) indica visibilidade privada. Mencionamos a ‘amizade’ (*friendship*) como uma exceção à visibilidade privada. A *friendship* é discutida no Capítulo 10.
- 9.2** b.
- 9.3** O projeto para classe Account produz o arquivo de cabeçalho na Figura 9.27.

9.13 Síntese

Este capítulo aprofundou nossa discussão sobre classes, utilizando um rico estudo de caso da classe Time para introduzir vários novos recursos de classes. Você viu que as funções-membro são normalmente menores que as funções globais porque podem acessar diretamente os membros de dados de um objeto; portanto, as funções-membro podem receber menos argumentos que as funções em linguagens de programação procedural. Você aprendeu a utilizar o operador seta para acessar os membros de um objeto por meio de um ponteiro do tipo da classe do objeto.

Você aprendeu que as funções-membro têm escopo de classe — isto é, o nome da função-membro só é conhecido por outros membros da classe a menos que referenciado por meio de um objeto da classe, uma referência a um objeto da classe, um ponteiro para um objeto da classe ou pelo operador de resolução de escopo binário. Também discutimos as funções de acesso (comumente utilizadas para recuperar os valores de membros de dados ou testar a verdade ou falsidade de condições) e funções utilitárias (funções-membro `private` que suportam a operação de funções-membro `public` da classe).

Você aprendeu que um construtor pode especificar argumentos-padrão que permitem que ele seja chamado de várias maneiras. Você também aprendeu que qualquer construtor que pode ser chamado sem argumentos é um construtor-padrão e que há, no máximo, um construtor-padrão por classe. Discutimos os destrutores e seus propósitos de realizar uma faxina de terminação em um objeto de uma classe antes de esse objeto ser destruído. Demonstramos também a ordem em que os construtores e destrutores de um objeto são chamados.

Demonstramos os problemas que podem ocorrer quando uma função-membro retorna uma referência a um membro de dados `private`, que quebra o encapsulamento da classe. Mostramos também que os objetos do mesmo tipo podem ser atribuídos uns aos outros

```

1 // Figura 9.27: Account.h
2 // Definição da classe Account. Representa uma conta bancária
3 #ifndef ACCOUNT_H
4 #define ACCOUNT_H
5
6 class Account
7 {
8 public:
9     bool validatePIN( int ); // o PIN especificado pelo usuário é correto?
10    double getAvailableBalance(); // retorna o saldo disponível
11    double getTotalBalance(); // retorna saldo total
12    void credit( double ); // adiciona um valor a Account
13    void debit( double ); // subtrai um valor de Account
14 private:
15    int accountNumber; // número da conta
16    int pin; // PIN para autenticação
17    double availableBalance; // fundos disponíveis para saque
18    double totalBalance; // fundos disponíveis + fundos esperando compensação
19 }; // fim da classe Account
20
21 #endif // ACCOUNT_H

```

Figura 9.27 O arquivo de cabeçalho da classe Account baseado nas figuras 9.20 e 9.21.

utilizando atribuição-padrão de membro a membro. Por fim, discutimos o benefício de utilizar a biblioteca de classes para aprimorar a velocidade com que o código pode ser criado e para aumentar a qualidade do software.

O Capítulo 10 apresenta recursos de classe adicionais. Demonstraremos como `const` pode ser utilizado para indicar que uma função-membro não modifica o objeto de uma classe. Você aprenderá a construir classes com composição — isto é, classes que contêm objetos de outras classes como membros. Mostraremos como uma classe pode permitir que as chamadas funções ‘friend’ (amigas) acessem membros não-`public` da classe. Também mostraremos como funções-membro não-`static` de uma classe podem utilizar um ponteiro especial chamado `this` para acessar os membros de um objeto. Em seguida, você aprenderá a utilizar os operadores C++ `new` e `delete`, que permitem aos programadores obter e liberar memória conforme necessário durante a execução de um programa.

Resumo

- As diretivas de pré-processador `#ifndef` (que quer dizer ‘se não definido’) e `#endif` são utilizadas para impedir múltiplas inclusões de um arquivo de cabeçalho. Se o código entre essas diretivas não tiver sido incluído anteriormente em um aplicativo, `#define` define um nome que pode ser utilizado para impedir inclusões futuras, e o código é incluído no arquivo de código-fonte.
- Os membros de dados de uma classe não podem ser inicializados no local em que são declarados no corpo de classe (exceto pelos membros de dados `static const` de uma classe do tipo inteiro ou do tipo `enum` como você verá no Capítulo 10). É altamente recomendado que esses membros de dados sejam inicializados pelo construtor da classe (uma vez que não há inicialização-padrão de membros de dados de tipos fundamentais).
- O manipulador de fluxo `setfill` especifica o caractere de preenchimento que é exibido quando um inteiro é enviado para a saída em um campo que é maior do que o número de dígitos no valor.
- Por padrão, os caracteres de preenchimento aparecem antes dos dígitos no número.
- O manipulador de fluxo `setfill` é uma configuração ‘aderente’, o quer dizer que, uma vez que o caractere de preenchimento é configurado, ele se aplica a todos os campos subsequentes sendo impressos.
- Embora uma função-membro declarada em uma definição de classe possa ser definida fora dessa definição de classe (e associada à classe via operador binário de resolução de escopo), essa função-membro ainda está dentro do escopo dessa classe; por exemplo, seu nome é conhecido somente pelos outros membros da classe a menos que referenciado via um objeto da classe, uma referência a um objeto da classe ou um ponteiro para um objeto da classe.
- Se uma função-membro é definida no corpo de uma definição de classe, o compilador C++ tenta colocar inline as chamadas para função-membro.
- As classes não precisam ser criadas ‘a partir do zero’. Em vez disso, elas podem incluir objetos de outras classes como membros ou podem ser derivadas de outras classes que fornecem atributos e comportamentos que podem ser utilizados pelas novas classes. Incluir objetos de uma classe como membros de outras classes é chamado de composição.
- Os membros de dados e as funções-membro de uma classe pertencem ao escopo dessa classe.
- As funções não-membro são definidas no escopo de arquivo.
- Dentro do escopo de uma classe, os membros da classe são imediatamente acessíveis por todas as funções-membro dessa classe e podem ser referenciados pelo nome.
- Fora do escopo de uma classe, os membros de classe são referenciados por um dos handles em um objeto — um nome de objeto, uma referência a um objeto ou um ponteiro para um objeto.
- As funções-membro de uma classe podem ser sobrecarregadas, mas apenas por outras funções-membro dessa classe.
- Para sobrecarregar uma função-membro, forneça na definição da classe um protótipo para cada versão da função sobrecarregada e uma definição separada para cada versão da função.
- As variáveis declaradas em uma função-membro têm escopo de bloco e são conhecidas somente por essa função.
- Se uma função-membro definir uma variável com o mesmo nome de uma variável com escopo de classe, a variável de escopo de classe é ocultada pela variável de escopo de bloco no escopo de bloco.
- O operador de seleção de membro ponto (`.`) é precedido pelo nome de um objeto ou por uma referência a um objeto para acessar membros `public` do objeto.
- O operador de seleção de membro seta (`->`) é precedido por um ponteiro para um objeto para acessar membros `public` desse objeto.
- Os arquivos de cabeçalho contêm algumas partes da implementação e dicas sobre outras. As funções-membro inline, por exemplo, precisam estar em um arquivo de cabeçalho, para que, quando o compilador compilar um cliente, este possa incluir adequadamente a definição de função `inline`.
- Os membros `private` de uma classe que são listados na definição de classe no arquivo de cabeçalho são visíveis aos clientes, mesmo que estes possam não acessar os membros `private`.

- Uma função utilitária (também chamada de função auxiliar) é uma função-membro `private` que suporta a operação de funções-membro `public` da classe. As funções utilitárias não foram projetadas para ser utilizadas por clientes de uma classe (mas podem ser utilizadas por `friends` de uma classe).
- Como outras funções, os construtores podem especificar argumentos-padrão.
- O destrutor de uma classe é chamado implicitamente quando um objeto da classe é destruído.
- O nome do destrutor de uma classe é o caractere til (`~`) seguido pelo nome da classe.
- Na realidade, um destrutor não libera o armazenamento de um objeto — ele realiza a faxina de terminação antes de o sistema reivindicar a memória de um objeto, assim a memória pode ser reutilizada para armazenar novos objetos.
- Um destrutor não recebe parâmetros nem retorna um valor. Uma classe pode ter somente um destrutor.
- Se o programador não fornecer um destrutor explicitamente, o compilador cria um destrutor ‘vazio’, então cada classe tem exatamente um destrutor.
- A ordem em que os construtores e destrutores são chamados depende da ordem em que a execução entra e sai dos escopos onde os objetos são instanciados.
- Geralmente, as chamadas de destrutor são feitas na ordem inversa das chamadas de construtor correspondentes, mas as classes de armazenamento de objetos podem alterar a ordem em que os destrutores são chamados.
- Uma referência a um objeto é um alias para o nome do objeto e, portanto, pode ser utilizada à esquerda de uma instrução de atribuição. Nesse contexto, a referência faz um *lvalue* perfeitamente aceitável que pode receber um valor. Uma maneira de utilizar essa capacidade (infelizmente!) é fazer uma função-membro `public` de uma classe retornar uma referência a um membro de dados `private` dessa classe. Se a função retorna uma referência `const`, então a referência não pode ser utilizada como um *lvalue* modificável.
- O operador de atribuição (`=`) pode ser utilizado para atribuir um objeto a outro objeto do mesmo tipo. Por padrão, essa atribuição é realizada pela atribuição de membro a membro — cada membro do objeto à direita do operador de atribuição é atribuído individualmente ao mesmo membro no objeto à esquerda do operador de atribuição.
- Os objetos podem ser passados como funções argumentos e podem ser retornados a partir de funções. Essa passagem e esse retorno são realizados utilizando a passagem por valor por padrão — uma cópia do objeto é passada ou retornada. Nesses casos, o C++ cria um novo objeto e utiliza um construtor de cópia para copiar os valores do objeto original para o novo objeto. Explicamos esses casos detalhadamente no Capítulo 11, “Sobrecarga de operadores; objetos string e array”.
- Para cada classe, o compilador fornece um construtor de cópia-padrão que copia cada membro do objeto original para o membro correspondente do novo objeto.
- Existem muitas bibliotecas de classe, e outras estão sendo desenvolvidas em todo o mundo.
- A reusabilidade de software acelera o desenvolvimento de softwares de alta qualidade e poderosos. O desenvolvimento rápido de aplicativos (*rapid applications development* – RAD) por meio dos mecanismos de componentes reutilizáveis tornou-se um importante campo.

Terminologia

<code>#define</code> , diretiva de pré-processador	construtor sobrecarregado	handle de referência em um objeto
<code>#endif</code> , diretiva de pré-processador	derivar uma classe da outra	handle em um objeto
<code>#ifndef</code> , diretiva de pré-processador	desenvolvimento rápido de aplicativos	handle implícito em um objeto
<code>abort</code> , função	(<i>rapid application development</i> – RAD)	herança
agregação	destrutor	inicializador
argumentos-padrão com construtores	empacotador de pré-processador	objeto sai do escopo
ativo de software	escopo de arquivo	operador de seleção de membro seta (<code>-></code>)
atribuição de membro a membro	escopo de classe	ordem em que construtores e destrutores são chamados
atribuição-padrão de membro a membro	<code>exit</code> , função	passar um objeto por valor
atribuindo objetos de classe	função auxiliar	preparação de terminação
bibliotecas de classe	função de acesso	procedure pura
caractere de preenchimento	função predicado	<code>setfill</code> , manipulador de fluxo parametrizado
código reentrante	função-membro sobrecarregada	til (<code>~</code>), caractere, em um nome de destrutor
componentes reutilizáveis	handle de nome em um objeto	
composição	handle de objeto	
construtor de cópia	handle de ponteiro em um objeto	

Exercícios de revisão

9.1 Preencha as lacunas em cada uma das seguintes sentenças:

- Os membros de classe são acessados via operador _____ em conjunto com o nome de um objeto (ou referência a um objeto) da classe ou via operador _____ em conjunto com um ponteiro para um objeto da classe.
- Os membros de classe especificados como _____ são acessíveis apenas às funções-membro da classe e `friends` da classe.
- Os membros de classe especificados como _____ são acessíveis em qualquer lugar que um objeto da classe esteja no escopo.
- A _____ pode ser utilizada para atribuir um objeto de uma classe a outro objeto da mesma classe.

9.2 Localize o(s) erro(s) em cada uma das seguintes seqüências e explique como corrigi-lo(s):

- a) Suponha que o seguinte protótipo é declarado na classe `Time`:

```
void ~Time( int );
```

- b) A seguinte definição é uma definição parcial da classe `Time`:

```
class Time
{
public:
// protótipos de função
private:
    int hour = 0;
    int minute = 0;
    int second = 0;
}; // fim da classe Time
```

- c) Suponha que o seguinte protótipo é declarado na classe `Employee`:

```
int Employee( const char *, const char * );
```

Respostas dos exercícios de revisão

- 9.1** a) ponto (`.`), seta (`->`). b) `private`. c) `public`. d) atribuição-padrão de membro a membro (realizada pelo operador de atribuição).
- 9.2** a) Erro: Os destrutores não têm permissão de retornar valores (nem mesmo de especificar um tipo de retorno) nem de aceitar argumentos.
Correção: Remova o tipo de retorno `void` e o parâmetro `int` da declaração.
- b) Erro: Os membros não podem ser explicitamente inicializados na definição de classe.
Correção: Remova a inicialização explícita da definição de classe e inicialize os membros de dados em um construtor.
- c) Erro: Os construtores não têm permissão de retornar valores.
Correção: Remova o tipo de retorno `int` da declaração.

Exercícios

9.3 Qual é o propósito do operador de resolução de escopo?

9.4 (*Aprimorando a classe `Time`*) Forneça um construtor que seja capaz de utilizar a hora atual da função `time()` — declarada no cabeçalho `<ctime>` da C++ Standard Library — para inicializar um objeto da classe `Time`.

9.5 (*Classe `Complex`*) Crie uma classe chamada `Complex` para realizar aritmética com números complexos. Escreva um programa para testar sua classe.

Os números complexos têm a forma

$$\text{parteReal} + \text{parteImaginária} * i$$

onde i é

$$\sqrt{-1}$$

Utilize as variáveis `double` para representar os dados `private` da classe. Forneça um construtor que permita que um objeto dessa classe seja inicializado quando ele for declarado. O construtor deve conter valores-padrão no caso de nenhum inicializador ser fornecido. Forneça as funções-membro `public` que realizam as tarefas a seguir:

- Somar dois números `Complex`: as partes reais são somadas de um lado e as partes imaginárias são somadas de outro.
- Subtrair dois números `Complex`: a parte real do operando direito é subtraída da parte real do operando esquerdo e a parte imaginária do operando direito é subtraída da parte imaginária do operando esquerdo.
- Imprimir os números `Complex` na forma (a, b) , onde a é a parte real e b é a parte imaginária.

- 9.6** (*Classe Rational*) Crie uma classe chamada `Rational` para realizar aritmética com frações. Escreva um programa para testar sua classe.
- Utilize variáveis do tipo inteiro para representar os dados `private` da classe — o numerador e o denominador. Forneça um construtor que permita que um objeto dessa classe seja inicializado quando ele for declarado. O construtor deve conter valores-padrão no caso de nenhum inicializador ser fornecido e deve armazenar a fração na forma reduzida. Por exemplo, a fração
- $$\frac{2}{4}$$
- seria armazenada no objeto como 1 no `numerator` e 2 no `denominator`. Forneça funções-membro `public` que realizam cada uma das tarefas a seguir:
- Somar dois números `Rational`. O resultado deve ser armazenado na forma reduzida.
 - Subtrair dois números `Rational`. O resultado deve ser armazenado na forma reduzida.
 - Multiplicar dois números `Rational`. O resultado deve ser armazenado na forma reduzida.
 - Dividir dois números `Rational`. O resultado deve ser armazenado na forma reduzida.
 - Imprimir os números `Rational` na forma `a/b`, onde `a` é o numerador e `b` é o denominador.
 - Imprimir os números `Rational` em formato de ponto flutuante.
- 9.7** (*Aprimorando a classe Time*) Modifique a classe `Time` das figuras 9.8–9.9 para incluir uma função-membro `tick` que incrementa a hora armazenada em um objeto `Time` por um segundo. O objeto `Time` sempre deve permanecer em um estado consistente. Escreva um programa que testa a função-membro `tick` em um loop que imprime a hora no formato-padrão durante cada iteração do loop para ilustrar que a função-membro `tick` funciona corretamente. Certifique-se de testar os seguintes casos:
- Incrementar para o próximo minuto.
 - Incrementar para a próxima hora.
 - Incrementar para o próximo dia (isto é, 11:59:59 PM para 12:00:00 AM).
- 9.8** (*Aprimorando a classe Date*) Modifique a classe `Date` das figuras 9.17–9.18 para realizar a verificação de erros nos valores inicializadores para membros de dados `month`, `day` e `year`. Além disso, forneça uma função-membro `nextDay` para incrementar o dia por um. O objeto `Date` sempre deve permanecer em um estado consistente. Escreva um programa que testa a função `nextDay` em um loop que imprime a data durante cada iteração para ilustrar que `nextDay` funciona corretamente. Certifique-se de testar os seguintes casos:
- Incrementar para o próximo mês.
 - Incrementar para o próximo ano.
- 9.9** (*Combinando a classe Time e a classe Date*) Combine a classe `Time` modificada do Exercício 9.7 e a classe `Date` modificada do Exercício 9.8 em uma classe chamada `DateAndTime`. (No Capítulo 12, discutiremos a herança, que permitirá realizar essa tarefa rapidamente sem modificar as definições de classe existentes.) Modifique a função `tick` para chamar a função `nextDay` se a hora for incrementada para o dia seguinte. Modifique as funções `printStandard` e `printUniversal` para gerar saída da data e da hora. Escreva um programa para testar a nova classe `DateAndTime`. Especificamente, teste incrementar a hora para o dia seguinte.
- 9.10** (*Retornando indicadores de erros das funções set da classe Time*) Modifique as funções `set` na classe `Time` das figuras 9.8–9.9 para retornar valores de erros apropriados se uma tentativa de *configurar* um membro de dados de um objeto da classe `Time` para um valor inválido for feita. Escreva um programa que testa sua nova versão da classe `Time`. Exiba mensagens de erro quando funções `set` retornarem valores de erros.
- 9.11** (*Classe Rectangle*) Crie uma classe `Rectangle` com atributos `length` e `width`, cada um dos quais assume o padrão de 1. Forneça funções-membro que calculam os atributos `perimeter` e `area` do retângulo. Além disso, forneça as funções `set` e `get` para os atributos `length` e `width`. As funções `set` devem verificar se `length` e `width` são números de ponto flutuante maiores que 0,0 e menores que 20,0.
- 9.12** (*Classe Rectangle aprimorada*) Crie uma classe `Rectangle` mais sofisticada do que aquela que você criou no Exercício 9.11. Essa classe armazena somente as coordenadas cartesianas dos quatro vértices do retângulo. O construtor chama uma função `set` que aceita quatro conjuntos de coordenadas e verifica se cada um deles está no primeiro quadrante sem coordenadas `x` ou `y` individualmente maiores que 20,0. A função `set` também verifica se as coordenadas fornecidas especificam de fato um retângulo. Forneça funções-membro que calculam `length`, `width`, `perimeter` e `area`. O comprimento é o maior das duas dimensões. Inclua uma função predicado `square` que determina se o retângulo é um quadrado.
- 9.13** (*Aprimorando a classe Rectangle*) Modifique a classe `Rectangle` do Exercício 9.12 para incluir uma função `draw` que exhibe o retângulo dentro de uma caixa de 25 por 25 para incluir a parte do primeiro quadrante em que o retângulo reside. Inclua uma função `setFillCharacter` para especificar o caractere a partir do qual o corpo do retângulo será desenhado. Inclua uma função `setPerimeterCharacter` para especificar o caractere que será utilizado para desenhar a borda do retângulo. Se você se sentir ambicioso, talvez queira incluir funções para dimensionar o tamanho do retângulo, rotacioná-lo e movê-lo dentro da parte designada do primeiro quadrante.
- 9.14** (*Classe HugeInteger*) Crie uma classe `HugeInteger` que utiliza um array de 40 elementos de dígitos para armazenar inteiros com até 40 dígitos cada. Forneça as funções-membro `input`, `output`, `add` e `subtract`. Para comparar objetos `HugeInteger`, forneça funções `isEqualTo`, `isNotEqualTo`, `isGreaterThan`, `isLessThan`, `isGreaterThanOrEqualTo` e `isLessThanOrEqualTo` — cada uma

dessas é uma função ‘predicado’ que simplesmente retorna `true` se o relacionamento se mantém entre os dois `HugeIntegers`, e retorna `false` se o relacionamento não se mantém. Além disso, forneça uma função predicado `isZero`. Se estiver motivado, forneça as funções-membro `multiply`, `divide` e `modulus`.

- 9.15** (*Classe `TicTacToe`*) Crie uma classe `TicTacToe` que permitirá escrever um programa completo para jogar o jogo-da-velha (Tic-Tac-Toe). A classe contém como dados `private` um array bidimensional 3 por 3 de inteiros. O construtor deve inicializar a grade vazia com todos como zero. Permita dois jogadores humanos. Para onde quer que o primeiro jogador se mova, coloque um 1 no quadrado especificado. Coloque 2 para onde quer que o segundo jogador se mova. Todo movimento deve ocorrer em um quadrado vazio. Depois de cada movimento, determine se houve uma derrota ou um empate. Se você se sentir motivado, modifique seu programa de modo que o computador faça o movimento para um dos jogadores. Além disso, permita que o jogador especifique se quer ser o primeiro ou o segundo. Se você se sentir excepcionalmente motivado, desenvolva um programa que jogue um jogo-da-velha tridimensional em uma grade 4 por 4 por 4. [Atenção: Esse é um projeto extremamente desafiador que pode exigir muitas semanas de esforço!]



*Mas o que, para servir nossos
fins pessoais,
Nos proíbe de enganar nossos
amigos?*

Charles Churchill

*Em vez dessa divisão absurda
em sexos, deveríamos classificar
as pessoas como estáticas e
dinâmicas.*

Evelyn Waugh

Não tenha amigos iguais a você.
Confúcio

Classes: um exame mais profundo, parte 2

OBJETIVOS

Neste capítulo, você aprenderá:

- A especificar objetos const (constante) e funções-membro const.
- A criar objetos compostos de outros objetos.
- A utilizar funções friend e classes friend.
- A utilizar o ponteiro this.
- A criar e destruir dinamicamente objetos com os operadores new e delete, respectivamente.
- A utilizar membros de dados e funções-membro static.
- O conceito de uma classe contêiner.
- A noção de classes iteradoras que percorrem os elementos de classes contêineres.
- A utilizar classes proxy para ocultar detalhes de implementação dos clientes de uma classe.

- 10.1** Introdução
- 10.2** Objetos `const` (constante) e funções-membro `const`
- 10.3** Composição: objetos como membros de classes
- 10.4** Funções `friend` e classes `friend`
- 10.5** Utilizando o ponteiro `this`
- 10.6** Gerenciamento de memória dinâmico com os operadores `new` e `delete`
- 10.7** Membros de classe `static`
- 10.8** Abstração de dados e ocultamento de informações
 - 10.8.1** Exemplo: tipo de dados abstrato `array`
 - 10.8.2** Exemplo: tipo de dados abstrato `string`
 - 10.8.3** Exemplo: tipo de dados abstrato `queue`
- 10.9** Classes contêineres e iteradores
- 10.10** Classes proxy
- 10.11** Síntese

Resumo | Terminologia | Exercícios de revisão | Respostas dos exercícios de revisão | Exercícios

10.1 Introdução

Neste capítulo, continuamos nosso estudo de classes e abstração de dados com vários tópicos mais avançados. Utilizamos objetos `const` e funções-membro `const` para impedir modificações de objetos e impor o princípio do menor privilégio. Discutimos a composição — uma forma de reutilização em que uma classe pode ter objetos de outras classes como membros. Em seguida, introduzimos a amizade (*friendship*), que permite ao designer de classes especificar funções não-membro que podem acessar membros não-`public` da classe — uma técnica que é freqüentemente utilizada na sobrecarga de operadores (Capítulo 11) por razões de desempenho. Discutimos um ponteiro especial (chamado `this`), que é um argumento implícito para cada uma das funções-membro não-`static` de uma classe que permite que essas funções-membro acessem os membros de dados corretos e outras funções-membro não-`static` do objeto. Então, discutimos o gerenciamento de memória dinâmico e mostramos como criar e destruir objetos dinamicamente com os operadores `new` e `delete`. Em seguida, motivamos a necessidade de membros de classe `static` e mostramos como utilizar os membros de dados e funções-membro `static` em suas próprias classes. Por fim, mostramos como criar uma classe proxy para ocultar os detalhes de implementação de uma classe (incluindo seus membros de dados `private`) a partir de clientes da classe.

Lembre-se de que o Capítulo 3 introduziu a classe `string` da C++ Standard Library para representar strings como objetos de classe completos. Neste capítulo, porém, utilizamos as strings baseadas em ponteiro introduzidas no Capítulo 8 para ajudar o leitor a dominar ponteiros e a se preparar para o mundo profissional em que verá uma grande quantidade de código C legado implementado nas duas últimas décadas. Assim, o leitor se familiarizará com os dois métodos mais predominantes de criar e manipular strings em C++.

10.2 Objetos `const` (constante) e funções-membro `const`

Enfatizamos o princípio do menor privilégio como um dos princípios mais fundamentais da boa engenharia de software. Vejamos como esse princípio se aplica aos objetos.

Alguns objetos precisam ser modificáveis e alguns não. O programador pode utilizar a palavra-chave `const` para especificar que um objeto não é modificável e que qualquer tentativa de modificá-lo deve resultar em um erro de compilação. A instrução

```
const Time noon( 12, 0, 0 );
```

declara um objeto `const noon` da classe `Time` e o inicializa como 12:00 AM (meio-dia).



Observação de engenharia de software 10.1

Declarar um objeto como `const` ajuda a impor o princípio do menor privilégio. As tentativas de modificar o objeto são capturadas em tempo de compilação em vez de causar erros em tempo de execução. Utilizar `const` adequadamente é crucial para o design de classe, o design de programa e a codificação adequados.



Dica de desempenho 10.1

Declarar variáveis e objetos `const` pode melhorar o desempenho — os sofisticados compiladores de otimização atuais podem realizar certas otimizações em constantes que não podem ser executadas em variáveis.

Os compiladores C++ não permitem chamadas de função-membro para objetos `const` a menos que as próprias funções-membro também sejam declaradas `const`. Isso é verdadeiro mesmo para funções-membro `get` que não modificam o objeto. Além disso, o compilador não permite que funções-membro declaradas `const` modifiquem o objeto.

Uma função é especificada como `const` *tanto* em seu protótipo (Figura 10.1; linhas 19–24) *como* em sua definição (Figura 10.2; linhas 47, 53, 59 e 65) inserindo a palavra-chave `const` depois da lista de parâmetros da função e, no caso da definição da função, antes da chave esquerda que inicia o corpo da função.



Erro comum de programação 10.1

Definir como `const` uma função-membro que modifica um membro de dados de um objeto é um erro de compilação.



Erro comum de programação 10.2

Definir como `const` uma função-membro que chama uma função-membro não-`const` da classe na mesma instância da classe é um erro de compilação.



Erro comum de programação 10.3

Invocar uma função-membro não-`const` em um objeto `const` é um erro de compilação.



Observação de engenharia de software 10.2

Uma função-membro `const` pode ser sobrecarregada com uma versão não-`const`. O compilador escolhe que função-membro sobrecarregada utilizar com base no objeto em que a função é invocada. Se o objeto for `const`, o compilador utilizará a versão `const`. Se o objeto não for `const`, o compilador utiliza a versão não-`const`.

Um problema interessante surge para construtores e destrutores, cada um dos quais em geral modifica objetos. A declaração `const` não é permitida para construtores e destrutores. Um construtor deve ter permissão de modificar um objeto para que o objeto possa ser inicializado adequadamente. Um destrutor deve ser capaz de realizar suas tarefas de faxina de terminação antes de a memória de um objeto ser reivindicada pelo sistema.



Erro comum de programação 10.4

Tentar declarar um construtor ou um destrutor `const` é um erro de compilação.

Definindo e utilizando funções-membro `const`

O programa das figuras 10.1–10.3 modifica a classe `Time` das figuras 9.9–9.10 fazendo suas funções `get` e `printUniversal` se tornarem `const`. No arquivo de cabeçalho `Time.h` (Figura 10.1), as linhas 19–21 e 24 agora incluem a palavra-chave `const` depois da lista de parâmetros de cada função. A definição correspondente de cada função na Figura 10.2 (linhas 47, 53, 59 e 65, respectivamente) também especifica a palavra-chave `const` depois da lista de parâmetros de cada função.

A Figura 10.3 instancia dois objetos `Time` — o objeto não-`const` `wakeUp` (linha 7) e o objeto `const` `noon` (linha 8). O programa tenta invocar as funções-membro não-`const` `setHour` (linha 13) e `printStandard` (linha 20) no objeto `const` `noon`. Em cada caso, o compilador gera uma mensagem de erro. O programa também ilustra as três outras combinações de chamada de função-membro em objetos — uma função-membro não-`const` em um objeto não-`const` (linha 11), uma função-membro `const` em um objeto não-`const` (linha 15) e uma função-membro `const` em um objeto `const` (linhas 17–18). As mensagens de erro geradas para chamadas de função-membro não-`const` em um objeto `const` são mostradas na janela de saída. Note que, embora alguns compiladores atuais emitam somente mensagens de advertência para as linhas 13 e 20 (permitindo, assim, que esse programa seja executado), consideramos esses avisos como erros — o padrão ANSI/ISO C++ não permite a invocação de uma função-membro não-`const` em um objeto `const`.

Note que, mesmo que um construtor precise ser uma função-membro não-`const` (Figura 10.2, linhas 15–18), ele ainda pode ser utilizado para inicializar um objeto `const` (Figura 10.3, linha 8). A definição do construtor `Time` (Figura 10.2, linhas 15–18) mostra que o construtor `Time` chama outra função-membro não-`const` — `setTime` (linhas 21–26) — para realizar a inicialização de um objeto `Time`. É permitido invocar uma função-membro não-`const` a partir da chamada de construtor como parte da inicialização de um objeto `const`. A ‘constância’ de um objeto `const` é imposta a partir do momento em que o construtor completa a inicialização do objeto até o destrutor desse objeto ser chamado.

Também note que a linha 20 na Figura 10.3 gera um erro de compilação mesmo se a função-membro `printStandard` da classe `Time` não modificar o objeto em que ela é invocada. O fato de uma função-membro não modificar um objeto não é suficiente para indicar que a função é uma função constante — a função deve ser explicitamente declarada `const`.

```

1 // Figura 10.1: Time.h
2 // Definição da classe Time.
3 // Funções-membro definidas em Time.cpp.
4 #ifndef TIME_H
5 #define TIME_H
6
7 class Time
8 {
9 public:
10     Time( int = 0, int = 0, int = 0 ); // construtor-padrão
11
12     // funções set
13     void setTime( int, int, int ); // configura time
14     void setHour( int ); // configura hour
15     void setMinute( int ); // configura minute
16     void setSecond( int ); // configura second
17
18     // funções get (normalmente declaradas const)
19     int getHour() const; // retorna hour
20     int getMinute() const; // retorna minute
21     int getSecond() const; // retorna second
22
23     // funções print (normalmente declaradas const)
24     void printUniversal() const; // imprime hora universal
25     void printStandard(); // imprime hora-padrão (deve ser const)
26 private:
27     int hour; // 0 - 23 (formato de relógio de 24 horas)
28     int minute; // 0 - 59
29     int second; // 0 - 59
30 }; // fim da classe Time
31
32 #endif

```

Figura 10.1 A definição da classe Time com funções-membro const.

```

1 // Figura 10.2: Time.cpp
2 // Definições de função-membro para a classe Time.
3 #include <iostream>
4 using std::cout;
5
6 #include <iomanip>
7 using std::setfill;
8 using std::setw;
9
10 #include "Time.h" // inclui definição da classe Time
11
12 // função constructor para inicializar dados private;
13 // chama a função-membro setTime para configurar variáveis;
14 // valores-padrão são 0 (ver definição da classe)
15 Time::Time( int hour, int minute, int second )
16 {
17     setTime( hour, minute, second );
18 } // fim do construtor de Time
19
20 // configura valores de hour, minute e second

```

Figura 10.2 Definições de função-membro da classe Time, incluindo as funções-membro const.

(continua)


```

21 void Time::setTime( int hour, int minute, int second )
22 {
23     setHour( hour );
24     setMinute( minute );
25     setSecond( second );
26 } // fim da função setTime
27
28 // configura valor de hour
29 void Time::setHour( int h )
30 {
31     hour = ( h >= 0 && h < 24 ) ? h : 0; // valida horas
32 } // fim da função setHour
33
34 // configura valor de minute
35 void Time::setMinute( int m )
36 {
37     minute = ( m >= 0 && m < 60 ) ? m : 0; // valida minutos
38 } // fim da função setMinute
39
40 // configura valor de second
41 void Time::setSecond( int s )
42 {
43     second = ( s >= 0 && s < 60 ) ? s : 0; // valida segundos
44 } // fim da função setSecond
45
46 // retorna valor de hour
47 int Time::getHour() const // funções get devem ser const
48 {
49     return hour;
50 } // fim da função getHour
51
52 // retorna valor de minute
53 int Time::getMinute() const
54 {
55     return minute;
56 } // fim da função getMinute
57
58 // retorna valor de second
59 int Time::getSecond() const
60 {
61     return second;
62 } // fim da função getSecond
63
64 // imprime a hora no formato universal de data/hora (HH:MM:SS)
65 void Time::printUniversal() const
66 {
67     cout << setfill( '0' ) << setw( 2 ) << hour << ":"
68         << setw( 2 ) << minute << ":" << setw( 2 ) << second;
69 } // fim da função printUniversal
70
71 // imprime a hora no formato-padrão de data/hora (HH:MM:SS AM ou PM)
72 void Time::printStandard() // nota a falta da declaração const
73 {
74     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
75         << ":" << setfill( '0' ) << setw( 2 ) << minute
76         << ":" << setw( 2 ) << second << ( hour < 12 ? " AM" : " PM" );
77 } // fim da função printStandard

```

Figura 10.2 Definições de função-membro da classe Time, incluindo as funções-membro const.

(continuação)

```

1 // Figura 10.3: fig10_03.cpp
2 // Tentando acessar um objeto const com funções-membro não-const.
3 #include "Time.h" // inclui definição da classe Time
4
5 int main()
6 {
7     Time wakeUp( 6, 45, 0 ); // objeto não-constante
8     const Time noon( 12, 0, 0 ); // objeto constante
9
10                                     // OBJETO      FUNÇÃO-MEMBRO
11     wakeUp.setHour( 18 ); // não-const não-const
12
13     noon.setHour( 12 ); // const non-const
14
15     wakeUp.getHour(); // não-const const
16
17     noon.getMinute(); // const const
18     noon.printUniversal(); // const const
19
20     noon.printStandard(); // const non-const
21     return 0;
22 } // fim de main

```

Mensagens de erro do compilador de linha de comando Borland C++:

```

Warning W8037 fig10_03.cpp 13: Non-const function Time::setHour(int)
called for const object in function main()
Warning W8037 fig10_03.cpp 20: Non-const function Time::printStandard()
called for const object in function main()

```

Mensagens de erro do compilador Microsoft Visual C++.NET:

```

C:\cpphtp5_examples\ch10\Fig10_01_03\fig10_03.cpp(13) : error C2662:
'Time::setHour' : cannot convert 'this' pointer from 'const Time' to
'Time &'
Conversion loses qualifiers
C:\cpphtp5_examples\ch10\Fig10_01_03\fig10_03.cpp(20) : error C2662:
'Time::printStandard' : cannot convert 'this' pointer from 'const Time' to
'Time &'
Conversion loses qualifiers

```

Mensagens de erro do compilador GNU C++:

```

fig10_03.cpp:13: error: passing `const Time' as `this' argument of
`void Time::setHour(int)' discards qualifiers
fig10_03.cpp:20: error: passing `const Time' as `this' argument of
`void Time::printStandard()' discards qualifiers

```

Figura 10.3 Objetos const e funções-membro const.

Inicializando um membro de dados **const** com um inicializador de membro

O programa das figuras 10.4–10.6 introduz a utilização da **sintaxe de inicializador de membro**. Todos os membros de dados *podem* ser inicializados utilizando a sintaxe de inicializador de membro, mas os membros de dados **const** e os membros de dados que são referências *devem* ser inicializados com os inicializadores de membro. Mais adiante neste capítulo, veremos que os objetos-membro também devem ser inicializados dessa maneira. Quando estudarmos herança no Capítulo 12, veremos que partes de classes básicas de classes derivadas também devem ser inicializadas dessa maneira.

A definição de construtor (Figura 10.5, linhas 11–16) utiliza uma **lista de inicializadores de membro** para inicializar os membros de dados da classe Increment — o inteiro não-const count e o inteiro const increment (declarados nas linhas 19–20 da Figura 10.4). Os inicializadores de membro aparecem entre a lista de parâmetros de um construtor e a chave esquerda que inicia o corpo do construtor. A lista de inicializadores de membro (Figura 10.5, linhas 12–13) é separada da lista de parâmetros por dois-pontos (:). Cada inicializador de membro consiste no nome do membro de dados seguido por parênteses contendo o valor inicial do membro. Nesse exemplo, count é inicializado com o valor de parâmetro de construtor c e increment é inicializado com o valor de parâmetro de construtor i. Observe que múltiplos inicializadores de membro são separados por vírgulas. Além disso, observe que a lista de inicializadores de membro executa antes de o corpo do construtor executar.



Observação de engenharia de software 10.3

Um objeto const não pode ser modificado por atribuição, portanto ele deve ser inicializado. Quando um membro de dados de uma classe é declarado const, um inicializador de membro deve ser utilizado para fornecer ao construtor o valor inicial do membro de dados para um objeto da classe. O mesmo é verdadeiro para referências.

```

1 // Figura 10.4: Increment.h
2 // Definição da classe Increment.
3 #ifndef INCREMENT_H
4 #define INCREMENT_H
5
6 class Increment
7 {
8 public:
9     Increment( int c = 0, int i = 1 ); // construtor-padrão
10
11     // definição da função addIncrement
12     void addIncrement()
13     {
14         count += increment;
15     } // fim da função addIncrement
16
17     void print() const; // imprime count e increment
18 private:
19     int count;
20     const int increment; // membro de dados const
21 }; // fim da classe Increment
22
23 #endif

```

Figura 10.4 A definição da classe Increment contendo o membro de dados não-const count e o membro de dados const increment.

```

1 // Figura 10.5: Increment.cpp
2 // Definições de função-membro para a classe Increment demonstram o uso do
3 // inicializador de membro para inicializar uma constante de um tipo de dados predefinido.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include "Increment.h" // inclui a definição da classe Increment
9
10 // construtor
11 Increment::Increment( int c, int i )
12 : count( c ), // inicializador para membro não-const
13   increment( i ) // inicializador requerido para membro const

```

Figura 10.5 O inicializador de membro utilizado para inicializar uma constante de um tipo de dados predefinido.

(continua)

```

14 {
15     // corpo vazio
16 } // fim do construtor Increment
17
18 // imprime valores de count e increment
19 void Increment::print() const
20 {
21     cout << "count = " << count << ", increment = " << increment << endl;
22 } // fim da função print

```

Figura 10.5 O inicializador de membro utilizado para inicializar uma constante de um tipo de dados predefinido.

(continuação)

```

1 // Figura 10.6: fig10_06.cpp
2 // Programa para testar a classe Increment.
3 #include <iostream>
4 using std::cout;
5
6 #include "Increment.h" // inclui a definição da classe Increment
7
8 int main()
9 {
10     Increment value( 10, 5 );
11
12     cout << "Before incrementing: ";
13     value.print();
14
15     for ( int j = 1; j <= 3; j++ )
16     {
17         value.addIncrement();
18         cout << "After increment " << j << ": ";
19         value.print();
20     } // fim do for
21
22     return 0;
23 } // fim de main

```

```

Before incrementing: count = 10, increment = 5
After increment 1: count = 15, increment = 5
After increment 2: count = 20, increment = 5
After increment 3: count = 25, increment = 5

```

Figura 10.6 Invocando funções-membro print e addIncrement de um objeto Increment.

*Tentando inicializar erroneamente um membro de dados **const** com uma atribuição*

O programa das figuras 10.7–10.9 ilustra os erros de compilação causados ao se tentar inicializar o membro de dados `const` `increment` com uma instrução de atribuição (Figura 10.8, linha 14) no corpo do construtor `Increment` em vez de inicializá-lo com um inicializador de membro. Observe que a linha 13 da Figura 10.8 não gera um erro de compilação, porque `count` não é declarado `const`. Observe também que os erros de compilação produzidos pelo Microsoft Visual C++.NET referem-se ao membro de dados `int` `increment` como um ‘objeto `const`’. O padrão ANSI/ISO C++ define um ‘objeto’ como qualquer ‘região de armazenamento’. Como as instâncias de classes, as variáveis do tipo fundamental também ocupam espaço na memória, portanto são freqüentemente referidas como ‘objetos’.



Erro comum de programação 10.5

Não fornecer um inicializador de membro para um membro de dados `const` é um erro de compilação.



Observação de engenharia de software 10.4

Membros de dados constantes (objetos const e variáveis const) e membros de dados declarados como referências devem ser inicializados com a sintaxe de inicializador de membro; não são permitidas atribuições para esses tipos de dados no corpo do construtor.

Observe que a função print (Figura 10.8, linhas 18–21) é declarada const. Pode parecer estranho rotular essa função const, porque, provavelmente, um programa nunca terá um objeto const Increment. Entretanto, é possível que um programa venha a ter uma referência const a um objeto Increment ou um ponteiro para const que aponta para um objeto Increment. Em geral, isso ocorre quando objetos da classe Increment são passados para funções ou retornados de funções. Nesses casos, somente as funções-membro const da classe Increment podem ser chamadas por meio da referência ou do ponteiro. Portanto, é razoável declarar a função print como const — fazer isso impede erros nas situações em que um objeto Increment é tratado como um objeto const.

```

1 // Figura 10.7: Increment.h
2 // Definição da classe Increment.
3 #ifndef INCREMENT_H
4 #define INCREMENT_H
5
6 class Increment
7 {
8 public:
9     Increment( int c = 0, int i = 1 ); // construtor-padrão
10
11     // definição da função addIncrement
12     void addIncrement()
13     {
14         count += increment;
15     } // fim da função addIncrement
16
17     void print() const; // imprime count e increment
18 private:
19     int count;
20     const int increment; // membro de dados const
21 }; // fim da classe Increment
22
23 #endif

```

Figura 10.7 A definição da classe Increment contendo o membro de dados não-const count e o membro de dados const increment.

```

1 // Figura 10.8: Increment.cpp
2 // Tentando inicializar uma constante de
3 // um tipo de dados predefinido com uma atribuição.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include "Increment.h" // inclui a definição da classe Increment
9
10 // construtor; membro constante 'increment' não é inicializado
11 Increment::Increment( int c, int i )
12 {
13     count = c; // permitida porque count não é constante
14     increment = i; // ERRO: Não é possível modificar um objeto const
15 } // fim do construtor Increment
16

```

Figura 10.8 Tentativa errônea de inicializar uma constante de um tipo de dados predefinido por atribuição.

(continua)

```

17 // imprime valores de count e increment
18 void Increment::print() const
19 {
20     cout << "count = " << count << ", increment = " << increment << endl;
21 } // fim da função print

```

Figura 10.8 Tentativa errônea de inicializar uma constante de um tipo de dados predefinido por atribuição.

(continuação)

```

1 // Figura 10.9: fig10_09.cpp
2 // Programa para testar a classe Increment.
3 #include <iostream>
4 using std::cout;
5
6 #include "Increment.h" // inclui a definição da classe Increment
7
8 int main()
9 {
10     Increment value( 10, 5 );
11
12     cout << "Before incrementing: ";
13     value.print();
14
15     for ( int j = 1; j <= 3; j++ )
16     {
17         value.addIncrement();
18         cout << "After increment " << j << ": ";
19         value.print();
20     } // fim do for
21
22     return 0;
23 } // fim de main

```

Mensagem de erro do compilador de linha de comando da Borland C++:

```

Error E2024 Increment.cpp 14: Cannot modify a const object in function
Increment::Increment(int,int)

```

Mensagens de erro do compilador Microsoft Visual C++.NET:

```

C:\cpptp5_examples\ch10\Fig10_07_09\Increment.cpp(12) : error C2758:
'Increment::increment' : must be initialized in constructor
base/member initializer list
C:\cpptp5_examples\ch10\Fig10_07_09\Increment.h(20) :
see declaration of 'Increment::increment'
C:\cpptp5_examples\ch10\Fig10_07_09\Increment.cpp(14) : error C2166:
l-value specifies const object

```

Mensagens de erro do compilador GNU C++:

```

Increment.cpp:12: error: uninitialized member 'Increment::increment' with
'const' type 'const int'
Increment.cpp:14: error: assignment of read-only data-member
`Increment::increment'

```

Figura 10.9 O programa para testar a classe `Increment` gera erros de compilação.



Dica de prevenção de erro 10.1

Declare como `const` todas as funções-membro de uma classe que não modificam o objeto em que elas operam. Ocasionalmente isso pode parecer inadequado, porque você não terá nenhuma intenção de criar objetos `const` dessa classe ou de acessar objetos dessa classe por meio de referências `const` ou ponteiros para `const`. Apesar disso, declarar essas funções-membro `const` oferece um benefício. Se a função-membro for inadvertidamente escrita para modificar o objeto, o compilador emitirá uma mensagem de erro.

10.3 Composição: objetos como membros de classes

Um objeto `AlarmClock` precisa saber quando deve soar seu alarme, então por que não incluir um objeto `Time` como um membro da classe `AlarmClock`? Essa capacidade é chamada **composição** e às vezes é referida como um **relacionamento tem um**. Uma classe pode ter objetos de outras classes como membros.



Observação de engenharia de software 10.5

Uma forma comum de reusabilidade de software é a composição, em que uma classe tem objetos de outras classes como membros.

Quando um objeto é criado, seu construtor é chamado automaticamente. Anteriormente, vimos como passar argumentos para o construtor de um objeto que criamos em `main`. Esta seção mostra como o construtor de um objeto pode passar argumentos para construtores de objeto-membro, o que é realizado via inicializadores de membro. Os objetos-membro são construídos na ordem em que são declarados na definição de classe (não na ordem em que são listados na lista de inicializadores de membro do construtor) e antes de os objetos da sua classe contêiner (às vezes chamados **objetos host**) serem construídos.

O programa das figuras 10.10–10.14 utiliza a classe `Date` (figuras 10.10–10.11) e a classe `Employee` (figuras 10.12–10.13) para demonstrar objetos como membros de outros objetos. A definição da classe `Employee` (Figura 10.12) contém membros de dados `private` `firstName`, `lastName`, `birthDate` e `hireDate`. Os membros `birthDate` e `hireDate` são objetos `const` da classe `Date`, que contém os membros de dados `private` `month`, `day` e `year`. O cabeçalho do construtor `Employee` (Figura 10.13, linhas 18–21) especifica que o construtor recebe quatro parâmetros (`first`, `last`, `dateOfBirth` e `dateOfHire`). Os primeiros dois parâmetros são utilizados no corpo do construtor para inicializar os arrays de caracteres `firstName` e `lastName`. Os dois últimos parâmetros são passados via inicializadores de membro para o construtor da classe `Date`. Dois-pontos (`:`) no cabeçalho separam os inicializadores de membro da lista de parâmetros. Os inicializadores de membro especificam os parâmetros do construtor `Employee` sendo passados para os construtores de objetos `Date` do membro. O parâmetro `dateOfBirth` é passado para o construtor do objeto `birthDate` (Figura 10.13, linha 20), e o parâmetro `dateOfHire` é passado para o construtor do objeto `hireDate` (Figura 10.13, linha 21). Novamente, os inicializadores de membro são separados por vírgulas. Ao estudar a classe `Date` (Figura 10.10), note que ela não fornece um construtor que recebe um parâmetro do tipo `Date`. Então,

```

1 // Figura 10.10: Date.h
2 // Definição da classe Date; funções-membro definidas em Date.cpp
3 #ifndef DATE_H
4 #define DATE_H
5
6 class Date
7 {
8 public:
9     Date( int = 1, int = 1, int = 1900 ); // construtor-padrão
10    void print() const; // imprime data no formato mês/dia/ano
11    ~Date(); // fornecida para confirmar a ordem de destruição
12 private:
13     int month; // 1-12 (janeiro-dezembro)
14     int day; // 1-31 conforme o mês
15     int year; // qualquer ano
16
17     // função utilitária para verificar se o dia é adequado para o mês e ano
18     int checkDay( int ) const;
19 }; // fim da classe Date
20
21 #endif

```

Figura 10.10 Definição da classe `Date`.

como a lista de inicializadores de membro no construtor da classe `Employee` é capaz de inicializar os objetos `birthDate` e `hireDate` passando os parâmetros do objeto `Date` para seus construtores de `Date`? Como mencionamos no Capítulo 9, o compilador fornece a cada classe um construtor de cópia padrão que copia cada membro do objeto de argumento do construtor para o membro correspondente do objeto sendo inicializado. O Capítulo 11 discute como os programadores podem definir construtores de cópia personalizados.

A Figura 10.14 cria dois objetos `Date` (linhas 11–12) e os passa como argumentos para o construtor do objeto `Employee` criado na linha 13. A linha 16 gera saída dos dados do objeto `Employee`. Quando cada objeto `Date` é criado nas linhas 11–12, o construtor `Date` definido nas linhas 11–28 da Figura 10.11 exibe uma linha de saída para mostrar que o construtor foi chamado (ver as duas primeiras linhas da saída de exemplo). [Nota: A linha 13 da Figura 10.14 produz duas chamadas adicionais ao construtor `Date` que não aparecem na saída do programa. Quando todos os objetos-membro `Date` de `Employee` forem inicializados na lista de inicializadores de membro do construtor `Employee`, o construtor de cópia padrão da classe `Date` é chamado. Esse construtor é definido implicitamente pelo compilador e não contém nenhuma instrução de saída para demonstrar quando ele é chamado. Discutimos os construtores de cópia e os construtores de cópia padrão em detalhes no Capítulo 11.]

A classe `Date` e a `Employee` incluem, cada uma, um destrutor (linhas 37–42 da Figura 10.11 e linhas 51–55 da Figura 10.13, respectivamente) que imprime uma mensagem quando um objeto de sua classe é destruído. Isso nos permite confirmar na saída de programa que objetos são construídos de dentro para fora e destruídos na ordem inversa de fora para dentro (isto é, os objetos-membro `Date` são destruídos depois do objeto `Employee` que os contém). Note as últimas quatro linhas na saída da Figura 10.14. As duas últimas linhas são as saídas do destrutor `Date` executando nos objetos `Date` `hire` (linha 12) e `birth` (linha 11), respectivamente. Essas saídas confirmam que os três objetos criados em `main` são destruídos na ordem inversa daquela em que foram construídos. (A saída do destrutor `Employee` é a quinta linha contando de cima para baixo.) A terceira e a quarta linhas contando de cima para baixo da janela de saída mostram os destrutores que executam para objetos-membro `hireDate` (Figura 10.12, linha 20) e `birthDate` (Figura 10.12, linha 19) do `Employee`.

```

1  // Figura 10.11: Date.cpp
2  // Definições de função-membro da classe Date.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Date.h" // inclui definição da classe Date
8
9  // construtor confirma valor adequado para month; chama
10 // função utilitária checkDay para confirmar o valor adequado para day
11 Date::Date( int mn, int dy, int yr )
12 {
13     if ( mn > 0 && mn <= 12 ) // valida month
14         month = mn;
15     else
16     {
17         month = 1; // mês inválido configurado como 1
18         cout << "Invalid month (" << mn << ") set to 1.\n";
19     } // fim de else
20
21     year = yr; // poderia validar yr
22     day = checkDay( dy ); // valida day
23
24     // gera saída do objeto Date para mostrar quando seu construtor é chamado
25     cout << "Date object constructor for date ";
26     print();
27     cout << endl;
28 } // fim do construtor Date
29
30 // imprime objeto Date na forma de mês/dia/ano
31 void Date::print() const
32 {
33     cout << month << '/' << day << '/' << year;
34 } // fim da função print

```

Figura 10.11 Definições de função-membro da classe `Date`.

(continua)

```

35
36 // gera saída do objeto Date para mostrar quando seu destrutor é chamado
37 Date::~Date()
38 {
39     cout << "Date object destructor for date ";
40     print();
41     cout << endl;
42 } // fim do destrutor ~Date
43
44 // função utilitária para confirmar valor adequado de day
45 // com base em month e year; também trata anos bissextos
46 int Date::checkDay( int testDay ) const
47 {
48     static const int daysPerMonth[ 13 ] =
49         { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
50
51     // determina se testDay é válido durante mês especificado
52     if ( testDay > 0 && testDay <= daysPerMonth[ month ] )
53         return testDay;
54
55     // verificação 29 de fevereiro para ano bissexto
56     if ( month == 2 && testDay == 29 && ( year % 400 == 0 ||
57         ( year % 4 == 0 && year % 100 != 0 ) ) )
58         return testDay;
59
60     cout << "Invalid day (" << testDay << ") set to 1.\n";
61     return 1; // deixa o objeto em estado consistente se valor ruim
62 } // fim da função checkDay

```

Figura 10.11 Definições de função-membro da classe Date.

(continuação)

```

1 // Figura 10.12: Employee.h
2 // Definição da classe Employee.
3 // Funções-membro definidas em Employee.cpp.
4 #ifndef EMPLOYEE_H
5 #define EMPLOYEE_H
6
7 #include "Date.h" // inclui definição da classe Date
8
9 class Employee
10 {
11 public:
12     Employee( const char * const, const char * const,
13         const Date &, const Date & );
14     void print() const;
15     ~Employee(); // fornecida para confirmar a ordem de destruição
16 private:
17     char firstName[ 25 ];
18     char lastName[ 25 ];
19     const Date birthDate; // composição: objeto-membro
20     const Date hireDate; // composição: objeto-membro
21 }; // fim da classe Employee
22
23 #endif

```

Figura 10.12 Definição da classe Employee para mostrar a composição.

```

1 // Figura 10.13: Employee.cpp
2 // Definições de função-membro da classe Employee.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // protótipos strlen e strncpy
8 using std::strlen;
9 using std::strncpy;
10
11 #include "Employee.h" // definição da classe Employee
12 #include "Date.h" // definição da classe Date
13
14 // construtor usa lista de inicializadores de membro para passar valores de inicializadores
15 // para construtores dos objetos-membro birthDate e hireDate
16 // [Nota: Isso invoca o chamado 'construtor de cópia padrão' que o
17 // compilador C++ fornece implicitamente.]
18 Employee::Employee( const char * const first, const char * const last,
19     const Date &dateOfBirth, const Date &dateOfHire )
20     : birthDate( dateOfBirth ), // inicializa birthDate
21       hireDate( dateOfHire ) // inicializa hireDate
22 {
23     // copia primeiro para firstName e verifica-se de que ele se ajusta
24     int length = strlen( first );
25     length = ( length < 25 ? length : 24 );
26     strncpy( firstName, first, length );
27     firstName[ length ] = '\0';
28
29     // copia por último para lastName e verifica-se de que ele se ajusta
30     length = strlen( last );
31     length = ( length < 25 ? length : 24 );
32     strncpy( lastName, last, length );
33     lastName[ length ] = '\0';
34
35     // gera saída do objeto Employee para mostrar quando o construtor é chamado
36     cout << "Employee object constructor: "
37          << firstName << " " << lastName << endl;
38 } // fim do construtor Employee
39
40 // imprime objeto Employee
41 void Employee::print() const
42 {
43     cout << lastName << ", " << firstName << " Hired: ";
44     hireDate.print();
45     cout << " Birthday: ";
46     birthDate.print();
47     cout << endl;
48 } // fim da função print
49
50 // gera saída do objeto Employee para mostrar quando seu destrutor é chamado
51 Employee::~Employee()
52 {
53     cout << "Employee object destructor: "
54          << lastName << ", " << firstName << endl;
55 } // fim do destrutor ~Employee

```

Figura 10.13 Definições de função-membro da classe Employee, incluindo o construtor com uma lista de inicializadores de membro.

Essas saídas confirmam que o objeto `Employee` foi destruído de fora para dentro — isto é, o destrutor `Employee` executa primeiro (saída mostrada na quinta linha contando de cima para baixo da janela de saída), então os objetos-membro são destruídos na ordem inversa daquela em que foram construídos. Novamente, as saídas na Figura 10.14 não mostraram os construtores que executam para esses objetos, porque esses eram os construtores de cópia padrão fornecidos pelo compilador C++.

Um objeto-membro não precisa ser explicitamente inicializado por um inicializador de membro. Se um inicializador de membro não for fornecido, o construtor-padrão do objeto-membro será chamado implicitamente. Os valores (se houver) estabelecidos pelo construtor-padrão podem ser sobrescritos pelas funções `set`. Entretanto, para inicialização complexa, essa abordagem pode exigir trabalho e tempo adicional significativos.



Erro comum de programação 10.6

Ocorre um erro de compilação se um objeto-membro não é inicializado com um inicializador de membro e se a classe do objeto-membro não fornece um construtor-padrão (isto é, a classe do objeto-membro define um ou mais construtores, mas nenhum deles é um construtor-padrão).

```

1 // Figura 10.14: fig10_14.cpp
2 // Demonstrando composição -- um objeto com objetos-membro.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Employee.h" // definição da classe Employee
8
9 int main()
10 {
11     Date birth( 7, 24, 1949 );
12     Date hire( 3, 12, 1988 );
13     Employee manager( "Bob", "Blue", birth, hire );
14
15     cout << endl;
16     manager.print();
17
18     cout << "\nTest Date constructor with invalid values:\n";
19     Date lastDayOff( 14, 35, 1994 ); // mês e dia inválidos
20     cout << endl;
21     return 0;
22 } // fim de main

```

```

Date object constructor for date 7/24/1949
Date object constructor for date 3/12/1988
Employee object constructor: Bob Blue

```

```
Blue, Bob Hired: 3/12/1988 Birthday: 7/24/1949
```

```

Test Date constructor with invalid values:
Invalid month (14) set to 1.
Invalid day (35) set to 1.
Date object constructor for date 1/1/1994

```

```

Date object destructor for date 1/1/1994
Employee object destructor: Blue, Bob
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949
Date object destructor for date 3/12/1988
Date object destructor for date 7/24/1949

```

Figura 10.14 Inicializadores de objeto-membro.



Dica de desempenho 10.2

Inicialize explicitamente objetos-membro por meio de inicializadores de membro. Isso elimina o overhead de ‘inicializar duplamente’ objetos-membro — uma vez quando o construtor-padrão do objeto-membro é chamado, e a outra quando as funções set são chamadas no corpo do construtor (ou posteriormente) para inicializar o objeto-membro.



Observação de engenharia de software 10.6

Se um membro de classe for um objeto de outra classe, tornar esse objeto-membro `public` não viola o encapsulamento e a ocultação de membros `private` desse objeto-membro. Entretanto, isso viola o encapsulamento e a ocultação da implementação da classe contêiner, portanto os objetos-membro dos tipos de classe ainda devem ser `private`, como todos os outros membros de dados.

Na linha 26 da Figura 10.11, note a chamada para a função-membro `Date::print`. Muitas funções-membro de classes em C++ não requerem argumentos. Isso porque cada função-membro contém um handle implícito (na forma de um ponteiro) para o objeto em que ela opera. Discutimos o ponteiro implícito, que é representado pela palavra-chave `this`, na Seção 10.5.

A classe `Employee` utiliza dois arrays de 25 caracteres (Figura 10.12, linhas 17–18) para representar o nome e sobrenome do `Employee`. Esses arrays podem desperdiçar espaço com nomes com menos de 24 caracteres. (Lembre-se, um caractere em cada array é para o caractere de terminação nulo, `'\0'`, da string.) Além disso, nomes longos com mais de 24 caracteres devem ser truncados para se ajustarem nesses arrays de caracteres de tamanho fixo. A Seção 10.7 apresenta outra versão da classe `Employee` que cria dinamicamente a quantidade exata de espaço necessária para armazenar o nome e o sobrenome.

Observe que a maneira mais simples de representar o nome e o sobrenome de um `Employee` utilizando a quantidade exata de espaço necessária é utilizar dois objetos `string` (a classe `string` C++ Standard Library foi introduzida no Capítulo 3). Se fizéssemos isso, o construtor `Employee` apareceria da seguinte maneira

```
Employee::Employee( const string &first, const string &last,
    const Date &dateOfBirth, const Date &dateOfHire )
: firstName( first ), // inicializa firstName
  lastName( last ), // inicializa lastName
  birthDate( dateOfBirth ), // inicializa birthDate
  hireDate( dateOfHire ) // inicializa hireDate
{
    // gera saída do objeto Employee para mostrar quando o construtor é chamado
    cout << "Employee object constructor: "
        << firstName << " " << lastName << endl;
} // fim do construtor Employee
```

Note que os membros de dados `firstName` e `lastName` (agora objetos `string`) são inicializados por inicializadores de membro. As classes `Employee` apresentadas nos capítulos 12–13 utilizam os objetos `string` dessa maneira. Neste capítulo, utilizamos strings baseadas em ponteiro para fornecer ao leitor contato adicional com a manipulação de ponteiro.

10.4 Funções friend e classes friend

Uma **função friend** de uma classe é definida fora do escopo dessa classe, ainda que tenha o direito de acessar membros não-`public` (e `public`) da classe. Funções independentes ou classes inteiras podem ser declaradas como amigas [*friends*] de outra classe.

Utilizar funções friend pode aprimorar o desempenho. Esta seção apresenta um exemplo mecânico de como uma função friend opera. Mais adiante no livro, as funções friend são utilizadas para sobrecarregar operadores no uso com objetos de classe (Capítulo 11) e para criar as classes iteradoras (Capítulo 21). Os objetos de uma classe iteradora podem selecionar itens sucessivamente ou realizar uma operação sobre itens em um objeto da classe contêiner (ver Seção 10.9). Os objetos de classes contêineres podem armazenar itens. Utilizar funções amigas [*friends*] é freqüentemente apropriado quando uma função-membro não pode ser utilizada para certas operações, como veremos no Capítulo 11.

Para declarar uma função como amiga de uma classe, preceda o protótipo de função na definição de classe com a palavra-chave `friend`. Para declarar todas as funções-membro da classe `ClassTwo` como amigas da classe `ClassOne`, coloque uma declaração na forma

```
friend class ClassTwo;
```

na definição da classe `ClassOne`.



Observação de engenharia de software 10.7

Mesmo que os protótipos para funções friend apareçam na definição de classe, as funções amigas não são funções-membro.



Observação de engenharia de software 10.8

As noções de acesso a membro de `private`, `protected` e `public` não são relevantes às declarações `friend`, portanto as declarações `friend` podem ser colocadas em qualquer lugar de uma definição de classe.



Boa prática de programação 10.1

Coloque todas as declarações de amizade em primeiro lugar dentro do corpo da definição de classe e não as preceda com nenhum especificador de acesso.

A amizade é concedida, não aceita — isto é, para a classe B ser amiga da classe A, esta deve declarar explicitamente que a classe B é sua amiga. Além disso, a relação de amizade não é nem simétrica nem transitiva; isto é, se a classe A é amiga da classe B e esta for amiga da classe C, você não pode inferir que a classe B seja amiga da classe A (novamente, a amizade não é simétrica), que a classe C seja amiga da classe B (também porque a amizade não é simétrica), ou que a classe A seja amiga da classe C (a amizade não é transitiva).



Observação de engenharia de software 10.9

Algumas pessoas na comunidade OOP acreditam que a 'amizade' corrompe o ocultamento de informações e enfraquece o valor da abordagem do projeto orientado a objetos. Neste texto, identificamos vários exemplos de uso responsável da amizade.

Modificando os dados **private** de uma classe com uma função **amiga**

A Figura 10.15 é um exemplo mecânico em que definimos a função `friend setX` para configurar o membro de dados `private x` da classe `Count`. Observe que a declaração `friend` (linha 10) aparece em primeiro lugar (por convenção) na definição de classe, mesmo antes de as funções-membro `public` serem declaradas. Novamente, essa declaração `friend` pode aparecer em qualquer lugar na classe.

A função `setX` (linhas 30–33) é uma função independente no estilo C — ela não é uma função-membro da classe `Count`. Por essa razão, quando `setX` é invocada para o objeto `counter`, a linha 42 passa `counter` como um argumento para `setX` em vez de utilizar um `handle` (como o nome do objeto) para chamar a função, como em

```
counter.setX( 8 );
```

Como mencionamos, a Figura 10.15 é um exemplo mecânico do uso da construção `friend`. Normalmente seria apropriado definir a função `setX` como uma função-membro da classe `Count`. Normalmente também seria apropriado separar o programa da Figura 10.15 em três arquivos:

1. Um arquivo de cabeçalho (por exemplo, `Count.h`) contendo a definição da classe `Count`, que por sua vez contém o protótipo de função `friend setX`
2. Um arquivo de implementação (por exemplo, `Count.cpp`) contendo as definições das funções-membro da classe `Count` e a definição da função `friend setX`
3. Um programa de teste (por exemplo, `fig10_15.cpp`) com a função `main`

Tentando modificar erroneamente um membro **private** com uma função não-**friend**

O programa da Figura 10.16 demonstra as mensagens de erro produzidas pelo compilador quando a função não-`friend` `cannotSetX` (linhas 29–32) é chamada para modificar o membro de dados `private x`.

É possível especificar funções sobrecarregadas como amigas de uma classe. Cada função sobrecarregada projetada para ser um amigo deve ser explicitamente declarada na definição de classe como uma amiga da classe.

```
1 // Figura 10.15: fig10_15.cpp
2 // Friends podem acessar membros private de uma classe.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 // definição da classe Count
8 class Count
9 {
10     friend void setX( Count &, int ); // declaração friend
11 public:
```

Figura 10.15 Friends podem acessar os membros `private` de uma classe.

(continua)

```

12     // construtor
13     Count()
14         : x( 0 ) // inicializa x como 0
15     {
16         // corpo vazio
17     } // fim do construtor de Count
18
19     // gera saída de x
20     void print() const
21     {
22         cout << x << endl;
23     } // fim da função print
24 private:
25     int x; // membro de dados
26 }; // fim da classe Count
27
28 // a função setX pode modificar os dados private de Count
29 // porque setX é declarada como uma amiga de Count (linha 10)
30 void setX( Count &c, int val )
31 {
32     c.x = val; // permitido pois setX Count é uma amiga de Count
33 } // fim da função setX
34
35 int main()
36 {
37     Count counter; // cria o objeto Count
38
39     cout << "counter.x after instantiation: ";
40     counter.print();
41
42     setX( counter, 8 ); // configura x utilizando uma função friend
43     cout << "counter.x after call to setX friend function: ";
44     counter.print();
45     return 0;
46 } // fim de main

```

counter.x after instantiation: 0
counter.x after call to setX friend function: 8

Figura 10.15 Friends podem acessar os membros private de uma classe.

(continuação)

```

1 // Figura 10.16: fig10_16.cpp
2 // Funções não-friend/não-membro não podem acessar dados private de uma classe.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 // definição da classe Count (observe que não há declaração de amizade)
8 class Count
9 {
10 public:
11     // construtor
12     Count()
13         : x( 0 ) // inicializa x como 0

```

Figura 10.16 As funções não-friend/não-membro não podem acessar membros private.

(continua)


```

14     {
15         // corpo vazio
16     } // fim do construtor de Count
17
18     // gera saída de x
19     void print() const
20     {
21         cout << x << endl;
22     } // fim da função print
23 private:
24     int x; // membro de dados
25 }; // fim da classe Count
26
27 // função cannotSetX tenta modificar dados private de Count,
28 // mas não pode porque a função não é amiga de Count
29 void cannotSetX( Count &c, int val )
30 {
31     c.x = val; // ERROR: não é possível acessar o membro private member em Count
32 } // fim da função cannotSetX
33
34 int main()
35 {
36     Count counter; // cria o objeto Count
37
38     cannotSetX( counter, 3 ); // cannotSetX não é friend
39     return 0;
40 } // fim de main

```

Mensagem de erro do compilador de linha de comando Borland C++:

```

Error E2247 Fig10_16/fig10_16.cpp 31: 'Count::x' is not accessible in
function cannotSetX(Count &,int)

```

Mensagens de erro do compilador Microsoft Visual C++.NET:

```

C:\cpphttp5_examples\ch10\Fig10_16\fig10_16.cpp(31) : error C2248: 'Count::x'
: cannot access private member declared in class 'Count'
C:\cpphttp5_examples\ch10\Fig10_16\fig10_16.cpp(24) : see declaration
of 'Count::x'
C:\cpphttp5_examples\ch10\Fig10_16\fig10_16.cpp(9) : see declaration
of 'Count'

```

Mensagens de erro do compilador GNU C++:

```

fig10_16.cpp:24: error: `int Count::x' is private
fig10_16.cpp:31: error: within this context

```

Figura 10.16 As funções não-friend/não-membro não podem acessar membros `private`.

(continuação)

10.5 Utilizando o ponteiro `this`

Vimos que as funções-membro de um objeto podem manipular os dados do objeto. Como as funções-membro sabem quais membros de dados do objeto devem manipular? Cada objeto tem acesso ao seu próprio endereço por um ponteiro chamado `this` (uma palavra-chave do C++). O ponteiro `this` de um objeto não faz parte do objeto em si — isto é, o tamanho da memória ocupado pelo ponteiro `this` não é refletido no resultado de uma operação `sizeof` no objeto. Em vez disso, o ponteiro `this` é passado (pelo compilador) como um argumento implícito para cada uma das funções-membro não-`static` do objeto. A Seção 10.7 introduz os membros de classe `static` e explica por que o ponteiro `this` não é passado implicitamente para funções-membro `static`.

Os objetos utilizam o ponteiro `this` implicitamente (como fizemos até agora) ou explicitamente para referenciar seus membros de dados e funções-membro. O tipo do ponteiro `this` depende do tipo do objeto e do fato de a função-membro em que `this` é usado ser ou não declarada como `const`. Por exemplo, em uma função-membro não constante da classe `Employee`, o ponteiro `this` tem o tipo `Employee * const` (um ponteiro constante para um objeto `Employee` não constante). Em uma função-membro constante da classe `Employee`, o ponteiro `this` tem o tipo de dados `const Employee * const` (um ponteiro constante para um objeto `Employee` constante).

Nosso primeiro exemplo nesta seção mostra a utilização implícita e explícita do ponteiro `this`; mais adiante neste capítulo e no Capítulo 11, mostramos alguns exemplos substanciais e sutis da utilização de `this`.

Utilizando o ponteiro `this` implícita e explicitamente para acessar membros de dados de um objeto

A Figura 10.17 demonstra a utilização implícita e explícita do ponteiro `this` para permitir que uma função-membro da classe `Test` imprima dados `private x` de um objeto `Test`.

Para fins de ilustração, a função-membro `print` (linhas 25–37) imprime primeiro `x` utilizando o ponteiro `this` implicitamente (linha 28) — apenas o nome do membro de dados é especificado. Então `print` utiliza duas notações diferentes para acessar `x` por meio do ponteiro `this` — o operador seta (`->`) com o ponteiro `this` (linha 32) e o operador ponto (`.`) com o ponteiro `this` desreferenciado (linha 36).

Observe os parênteses em torno de `*this` (linha 36) quando utilizado com o operador ponto de seleção de membro (`.`). Os parênteses são requeridos porque o operador ponto tem precedência mais alta que o operador `*`. Sem os parênteses, a expressão `*this.x` seria avaliada como se estivesse entre parênteses como `*( this.x )`, o que é um erro de compilação, porque o operador ponto não pode ser utilizado com um ponteiro.

Uma utilização interessante do ponteiro `this` é impedir que um objeto seja atribuído a si mesmo. Como veremos no Capítulo 11, a auto-atribuição pode causar sérios erros quando o objeto contiver ponteiros para armazenamento dinamicamente alocado.



Erro comum de programação 10.7

Tentar utilizar o operador de seleção de membro (`.`) com um ponteiro para um objeto é um erro de compilação — o operador ponto de seleção de membro pode ser utilizado apenas com um lvalue como o nome de um objeto, uma referência para um objeto ou um ponteiro desreferenciado para um objeto.

Utilizando o ponteiro `this` para permitir chamadas de função em cascata

Outra utilização do ponteiro `this` é permitir **chamadas de função-membro em cascata** nas quais múltiplas funções são invocadas na mesma instrução (como na linha 14 da Figura 10.20). O programa das figuras 10.18–10.20 modifica as funções `set` `setTime`, `setHour`, `setMinute` e `setSecond` da classe `Time` de modo que cada uma retorna uma referência a um objeto `Time` para permitir chamadas de função-membro em cascata. Note na Figura 10.19 que a última instrução no corpo de cada uma dessas funções-membro retorna `*this` (linhas 26, 33, 40 e 47) em um tipo de retorno de `Time &`.

O programa da Figura 10.20 cria objeto `Time t` (linha 11), então o utiliza em chamadas de função-membro em cascata (linhas 14 e 26). Por que a técnica de retornar `*this` como uma referência funciona? O operador ponto (`.`) associa da esquerda para a direita, então a linha 14 primeiro avalia `t.setHour( 18 )` e, em seguida, retorna uma referência ao objeto `t` como o valor dessa chamada de função. A expressão restante é então interpretada como

```
t.setMinute( 30 ).setSecond( 22 );
```

A chamada `t.setMinute( 30 )` executa e retorna uma referência ao objeto `t`. A expressão restante é interpretada como

```
t.setSecond( 22 );
```

A linha 26 também utiliza cascadeamento. As chamadas devem aparecer na ordem mostrada na linha 26, porque `printStandard`, como definido na classe, não retorna uma referência ao `t`. Colocar a chamada a `printStandard` antes da chamada a `setTime` na linha 26 resulta em erro de compilação. O Capítulo 11 apresenta vários exemplos práticos do uso de chamadas de função em cascata. Um tipo de exemplo assim utiliza múltiplos operadores `<<` com `cout` para gerar saída de múltiplos valores em uma única instrução.

10.6 Gerenciamento de memória dinâmico com os operadores `new` e `delete`

O C++ permite aos programadores controlar a alocação e desalocação de memória em um programa de qualquer tipo predefinido ou definido pelo usuário. Isso é conhecido como **gerenciamento de memória dinâmico** e é realizado com operadores `new` e `delete`. Lembre-se de que a classe `Employee` (figuras 10.12–10.13) utiliza dois arrays de 25 caracteres para representar o nome e sobrenome de um `Employee`. A definição da classe `Employee` (Figura 10.12) deve especificar o número de elementos em cada um desses arrays ao declará-los como membros de dados, porque o tamanho destes dita a quantidade de memória requerida para armazenar um objeto `Employee`. Como discutimos anteriormente, esses arrays podem desperdiçar espaço em nomes com menos de 24 caracteres. Além disso, os nomes com mais de 24 caracteres devem ser truncados para se ajustarem nesses arrays de tamanho fixo.

Não seria interessante se pudéssemos utilizar arrays com o número exato de elementos necessários para armazenar o nome e o sobrenome de um `Employee`? O gerenciamento de memória dinâmico permite fazer exatamente isso. Como veremos no exemplo da Seção 10.7, se substituirmos os membros de dados de array `firstName` e `lastName` por ponteiros para `char`, poderemos utilizar o operador `new` para **alocar** dinamicamente (isto é, reservar) a quantidade exata de memória necessária para armazenar todos os nomes em tempo de execução. Alocar memória dinamicamente desse modo faz com que um array (ou qualquer outro tipo predefinido ou definido pelo

usuário) seja criado no **armazenamento livre** (às vezes chamado **heap**) — uma região da memória atribuída a cada programa para armazenar objetos criados em tempo de execução. Uma vez que a memória de um array é alocada no armazenamento livre, podemos ganhar acesso a ela apontando um ponteiro para o primeiro elemento do array. Quando não precisamos mais dos arrays, podemos retornar a memória ao armazenamento livre utilizando o operador `delete` para **desalocar** (isto é, liberar) a memória, que então pode ser reutilizada por futuras operações `new`.

```

1 // Figura 10.17: fig10_17.cpp
2 // Utilizando o ponteiro this para referenciar membros de objeto.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 class Test
8 {
9 public:
10     Test( int = 0 ); // construtor-padrão
11     void print() const;
12 private:
13     int x;
14 }; // fim da classe Test
15
16 // construtor
17 Test::Test( int value )
18     : x( value ) // inicializa x como value
19 {
20     // corpo vazio
21 } // fim do construtor Test
22
23 // imprime x utilizando ponteiros this implícito e explícito;
24 // os parênteses em torno de *this são requeridos
25 void Test::print() const
26 {
27     // utiliza implicitamente o ponteiro this para acessar o membro x
28     cout << "          x = " << x;
29
30     // utiliza explicitamente o ponteiro this e o operador seta
31     // para acessar o membro x
32     cout << "\n this->x = " << this->x;
33
34     // utiliza explicitamente o ponteiro this desreferenciado e
35     // o operador ponto para acessar o membro x
36     cout << "\n(*this).x = " << ( *this ).x << endl;
37 } // fim da função print
38
39 int main()
40 {
41     Test testObject( 12 ); // instancia e inicializa testObject
42
43     testObject.print();
44     return 0;
45 } // fim de main

```

```

x = 12
this->x = 12
(*this).x = 12

```

Figura 10.17 Ponteiro `this` acessando implícita e explicitamente os membros de um objeto.

```

1 // Figura 10.18: Time.h
2 // Colocando chamadas de função-membro em cascata.
3
4 // Definição da classe Time.
5 // Funções-membro definidas em Time.cpp.
6 #ifndef TIME_H
7 #define TIME_H
8
9 class Time
10 {
11 public:
12     Time( int = 0, int = 0, int = 0 ); // construtor-padrão
13
14     // funções set (os tipos de retorno Time & que permitem cascadeamento)
15     Time &setTime( int, int, int ); // configura hour, minute, second
16     Time &setHour( int ); // configura hour
17     Time &setMinute( int ); // configura minute
18     Time &setSecond( int ); // configura second
19
20     // funções get (normalmente declaradas const)
21     int getHour() const; // retorna hour
22     int getMinute() const; // retorna minute
23     int getSecond() const; // retorna second
24
25     // funções print (normalmente declaradas const)
26     void printUniversal() const; // imprime hora universal
27     void printStandard() const; // imprime a hora-padrão
28 private:
29     int hour; // 0 - 23 (formato de relógio de 24 horas)
30     int minute; // 0 - 59
31     int second; // 0 - 59
32 }; // fim da classe Time
33
34 #endif

```

Figura 10.18 Definição da classe Time modificada para permitir chamadas de função-membro em cascata.

```

1 // Figura 10.19: Time.cpp
2 // Definições de função-membro para a classe Time.
3 #include <iostream>
4 using std::cout;
5
6 #include <iomanip>
7 using std::setfill;
8 using std::setw;
9
10 #include "Time.h" // definição da classe Time
11
12 // função constructor para inicializar dados private;
13 // chama a função-membro setTime para configurar variáveis;
14 // valores-padrão são 0 (ver definição de classe)
15 Time::Time( int hr, int min, int sec )
16 {
17     setTime( hr, min, sec );

```

Figura 10.19 Definições de função-membro da classe Time modificadas para permitir chamadas de função-membro em cascata.

(continua)

```

18 } // fim do construtor de Time
19
20 // configura valores de hour, minute e second
21 Time &Time::setTime( int h, int m, int s ) // observe o retorno Time &
22 {
23     setHour( h );
24     setMinute( m );
25     setSecond( s );
26     return *this; // permite cascadeamento
27 } // fim da função setTime
28
29 // configura valor de hour
30 Time &Time::setHour( int h ) // observe o retorno Time &
31 {
32     hour = ( h >= 0 && h < 24 ) ? h : 0; // valida horas
33     return *this; // permite cascadeamento
34 } // fim da função setHour
35
36 // configura valor de minute
37 Time &Time::setMinute( int m ) // observe o retorno Time &
38 {
39     minute = ( m >= 0 && m < 60 ) ? m : 0; // valida minutos
40     return *this; // permite cascadeamento
41 } // fim da função setMinute
42
43 // configura valor de second
44 Time &Time::setSecond( int s ) // observe o retorno Time &
45 {
46     second = ( s >= 0 && s < 60 ) ? s : 0; // valida segundos
47     return *this; // permite cascadeamento
48 } // fim da função setSecond
49
50 // obtém valor da hora
51 int Time::getHour() const
52 {
53     return hour;
54 } // fim da função getHour
55
56 // obtém valor dos minutos
57 int Time::getMinute() const
58 {
59     return minute;
60 } // fim da função getMinute
61
62 // obtém valor dos segundos
63 int Time::getSecond() const
64 {
65     return second;
66 } // fim da função getSecond
67
68 // imprime a hora no formato universal de data/hora (HH:MM:SS)
69 void Time::printUniversal() const
70 {
71     cout << setfill( '0' ) << setw( 2 ) << hour << ":"
72         << setw( 2 ) << minute << ":" << setw( 2 ) << second;
73 } // fim da função printUniversal
74

```

Figura 10.19 Definições de função-membro da classe Time modificadas para permitir chamadas de função-membro em cascata.

(continua)

```

75 // imprime a hora no formato-padrão de data/hora (HH:MM:SS AM ou PM)
76 void Time::printStandard() const
77 {
78     cout << ( ( hour == 0 || hour == 12 ) ? 12 : hour % 12 )
79         << ":" << setfill( '0' ) << setw( 2 ) << minute
80         << ":" << setw( 2 ) << second << ( hour < 12 ? " AM" : " PM" );
81 } // fim da função printStandard

```

Figura 10.19 Definições de função-membro da classe Time modificadas para permitir chamadas de função-membro em cascata. (continuação)

```

1 // Figura 10.20: fig10_20.cpp
2 // Colocando chamadas de função-membro em cascata com o ponteiro this.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Time.h" // definição da classe Time
8
9 int main()
10 {
11     Time t; // cria o objeto Time
12
13     // chamadas de função em cascata
14     t.setHour( 18 ).setMinute( 30 ).setSecond( 22 );
15
16     // gera saída da hora nos formatos universal e padrão
17     cout << "Universal time: ";
18     t.printUniversal();
19
20     cout << "\nStandard time: ";
21     t.printStandard();
22
23     cout << "\n\nNew standard time: ";
24
25     // chamadas de função em cascata
26     t.setTime( 20, 20, 20 ).printStandard();
27     cout << endl;
28     return 0;
29 } // fim de main

```

Universal time: 18:30:22

Standard time: 6:30:22 PM

New standard time: 8:20:20 PM

Figura 10.20 Colocando função-membro em cascata.

Novamente, apresentamos a classe Employee modificada como descrita aqui no exemplo da Seção 10.7. Primeiro, apresentamos os detalhes da utilização dos operadores new e delete para alocar memória dinamicamente para armazenamento de objetos, tipos fundamentais e arrays.

Considere a seguinte declaração e instrução:

```

Time *timePtr;
timePtr = new Time;

```

O operador `new` aloca armazenamento do tamanho adequado para um objeto do tipo `Time`, chama o construtor-padrão para inicializar o objeto e retorna um ponteiro do tipo especificado à direita do operador `new` (isto é, um `Time *`). Observe que `new` pode ser utilizado para alocar dinamicamente qualquer tipo fundamental (como `int` ou `double`) ou tipo de classe. Se `new` for incapaz de localizar espaço suficiente na memória para o objeto, ele indica que ocorreu um erro ‘lançando uma exceção’. O Capítulo 16, “Tratamento de exceções”, discute como lidar com as falhas de `new` no contexto do padrão ANSI/ISO C++. Em particular, mostraremos como ‘capturar’ a exceção lançada por `new` e tratá-la. Quando um programa não ‘captura’ uma exceção, ele termina imediatamente. [Nota: O operador `new` retorna um ponteiro 0 em versões do C++ anteriores ao padrão ANSI/ISO. Utilizamos a versão-padrão do operador `new` por todo este livro.]

Para destruir um objeto alocado dinamicamente e liberar o espaço para o objeto, utilize o operador `delete` da seguinte maneira:

```
delete timePtr;
```

Essa instrução chama primeiro o destrutor do objeto para o qual `timePtr` aponta e, então, desaloca a memória associada com o objeto. Depois da instrução precedente, a memória pode ser reutilizada pelo sistema para alocar outros objetos.



Erro comum de programação 10.8

Não liberar memória alocada dinamicamente quando ela não é mais necessária pode fazer com que o sistema fique sem memória prematuramente. Isso às vezes é chamado ‘vazamento de memória’.

O C++ permite fornecer um **inicializador** para uma variável do tipo fundamental recém-criada, como em

```
double *ptr = new double( 3.14159 );
```

que inicializa um `double` recém-criado como 3.14159 e atribui o ponteiro resultante a `ptr`. A mesma sintaxe pode ser utilizada para especificar uma lista de argumentos separados por vírgulas para o construtor de um objeto. Por exemplo,

```
Time *timePtr = new Time( 12, 45, 0 );
```

inicializa um objeto `Time` recém-criado como 12:45 PM e atribui o ponteiro resultante a `timePtr`.

Como discutido anteriormente, o operador `new` pode ser utilizado para alocar arrays dinamicamente. Por exemplo, um array de inteiros de 10 elementos pode ser alocado e atribuído a `gradesArray` como mostrado a seguir:

```
int *gradesArray = new int[ 10 ];
```

o que declara o ponteiro `gradesArray` e atribui a ele um ponteiro para o primeiro elemento de um array de 10 elementos alocado dinamicamente de inteiros. Lembre-se de que o tamanho de um array criado em tempo de compilação deve ser especificado utilizando uma expressão integral constante. Entretanto, o tamanho de um array alocado dinamicamente pode ser especificado com *qualquer* expressão integral que seja avaliada em tempo de execução. Observe também que, ao alocar dinamicamente um array de objetos, o programador não pode passar argumentos para o construtor de cada objeto. Em vez disso, cada objeto no array é inicializado por seu construtor-padrão. A fim de excluir o array alocado dinamicamente para o qual `gradesArray` aponta, utilize a instrução

```
delete [] gradesArray;
```

A instrução precedente desaloca o array para o qual `gradesArray` aponta. Se o ponteiro na instrução precedente apontar para um array de objetos, a instrução primeiro chamará o destrutor para cada objeto no array e, depois, desalocará a memória. Se a instrução precedente não incluísse os colchetes (`[]`) e `gradesArray` apontasse para um array de objetos, somente o primeiro objeto no array receberia uma chamada de destrutor.



Erro comum de programação 10.9

Utilizar `delete` em vez de `delete []` para arrays de objetos pode levar a erros de lógica em tempo de execução. Para assegurar que cada objeto no array recebe uma chamada de destrutor, sempre exclua a memória alocada como array com o operador `delete []`. De modo semelhante, sempre exclua a memória alocada como um elemento individual com o operador `delete`.

10.7 Membros de classe static

Há uma exceção importante à regra que diz que cada objeto de uma classe tem sua própria cópia de todos os membros de dados da classe. Em certos casos, apenas uma cópia de uma variável deve ser compartilhada por todos os objetos de uma classe. Um **membro de dados static** é utilizado por essas e outras razões. Essa variável representa informações no ‘nível da classe’ (isto é, uma propriedade da classe compartilhada por todas as instâncias, não uma propriedade de um objeto específico da classe). A declaração de um membro `static` inicia com a palavra-chave `static`. Lembre-se de que as versões da classe `GradeBook` no Capítulo 7 utilizam membros de dados `static` para armazenar constantes que representam o número de notas que todos os objetos `GradeBook` podem armazenar.

Vamos motivar ainda mais a necessidade de dados no nível da classe `static` com um exemplo. Suponha que tivéssemos um videogame com `Martians` [marcianos] e outras criaturas do espaço. Cada `Martian` tende a ser corajoso e a atacar outras criaturas espaciais quando o `Martian` está ciente de que há pelo menos cinco `Martians` presentes. Se menos de cinco estiverem presentes, cada `Martian` individualmente torna-se covarde. Assim, cada `Martian` precisa saber a `martianCount`. Poderíamos fornecer a cada instância da classe `Martian` um `martianCount` como um membro de dados. Nesse caso, cada `Martian` terá uma cópia separada do membro de dados. Toda vez que criarmos um novo `Martian`, teremos de atualizar o membro de dados `martianCount` em todos os objetos `Martian`. Isso exigiria

que cada objeto `Martian` tivesse, ou acessasse, handles para todos os outros objetos `Martians` na memória. Isso desperdiça espaço, com as cópias redundantes, e tempo, atualizando as cópias separadas. Em vez disso, declaramos que `martianCount` será `static`. Isso faz dados de escopo de classe `martianCount`. Todo `Martian` pode acessar `martianCount` como se fosse um membro de dados do `Martian`, mas apenas uma cópia da variável `static` `martianCount` é mantida pelo C++. Isso economiza espaço. Economizamos tempo fazendo o construtor `Martian` incrementar a variável `static` `martianCount` e o destrutor `Martian` decrementar `martianCount`. Como há somente uma cópia, não temos de incrementar ou decrementar cópias separadas de `martianCount` para cada objeto `Martian`.



Dica de desempenho 10.3

Utilize os membros de dados `static` para poupar armazenamento quando uma única cópia dos dados de todos os objetos de uma classe for suficiente.

Embora possam parecer variáveis globais, os membros de dados de uma classe `static` têm escopo de classe. Além disso, os membros `static` podem ser declarados `public`, `private` ou `protected`. Um membro de dados `static` do tipo fundamental é inicializado por padrão como 0. Se quiser um valor inicial diferente, um membro de dados `static` pode ser inicializado *uma vez* (e apenas uma única vez). Um membro de dados `const static` do tipo `int` ou `enum` pode ser inicializado em sua declaração na definição de classe. Entretanto, todos os outros membros de dados `static` devem ser definidos no escopo de arquivo (isto é, fora do corpo da definição de classe) e inicializados somente nessas definições. Observe que os membros de dados `static` de tipos de classe (isto é, objetos-membro `static`) que têm construtores-padrão não precisam ser inicializados porque seus construtores-padrão serão chamados.

Os membros `private` e `protected static` de uma classe são normalmente acessados por funções-membro `public` da classe ou por `friends` da classe. (No Capítulo 12, veremos que os membros `private` e `protected static` de uma classe também podem ser acessados por funções-membro `protected` da classe.) Os membros `static` de uma classe existem até mesmo quando não existe nenhum objeto dessa classe. Para acessar um membro de classe `public static` quando não existe nenhum objeto da classe, simplesmente prefixe o nome de classe e o operador de resolução de escopo binário (`::`) com o nome do membro de dados. Por exemplo, se nossa variável `martianCount` precedente é `public`, ela pode ser acessada com a expressão `Martian::martianCount` quando não houver nenhum objeto `Martian`. (Naturalmente, o uso de dados `public` não é encorajado.)

Os membros de classe `public static` de uma classe também podem ser acessados por qualquer objeto dessa classe utilizando o nome do objeto, o operador ponto e o nome do membro (por exemplo, `myMartian.martianCount`). Para acessar um membro de classe `private` ou `protected static` quando não houver nenhum objeto da classe, forneça uma função-membro `public static` e chame a função prefixando seu nome com o nome de classe e o operador de resolução de escopo binário. (Como veremos no Capítulo 12, uma função-membro `protected static` pode também servir a esse propósito.) Uma função-membro `static` é um serviço da classe, não de um objeto específico da classe.



Observação de engenharia de software 10.10

Os membros de dados `static` e funções-membro `static` de uma classe existem e podem ser utilizados mesmo quando nenhum objeto dessa classe tiver sido instanciado.

O programa das figuras 10.21–10.23 demonstra um membro de dados `private static` chamado `count` (Figura 10.21, linha 21) e uma função-membro `public static` chamada `getCount` (Figura 10.21, linha 15). Na Figura 10.22, a linha 14 define e inicializa o membro de dados `count` como zero no escopo de arquivo e as linhas 18–21 definem a função-membro `static` `getCount`. Note que nem a linha 14 nem a linha 18 incluem a palavra-chave `static`, ainda que ambas as linhas referenciem membros de classe `static`. Quando `static` é aplicado a um item no escopo de arquivo, esse item torna-se conhecido apenas nesse arquivo. Os membros `static` da classe precisam estar disponíveis a partir de qualquer código-cliente que acesse o arquivo, portanto não podemos declará-los `static` no arquivo `.cpp` — eles são declarados `static` apenas no arquivo `.h`. O membro de dados `count` mantém uma contagem do número de objetos da classe `Employee` que foram instanciados. Quando há objetos da classe `Employee`, o membro `count` pode ser referenciado por qualquer função-membro de um objeto `Employee` — na Figura 10.22, `count` é referenciado tanto pela linha 33 no construtor como pela linha 48 no destrutor. Além disso, observe que como `count` é um `int`, ele poderia ter sido inicializado no arquivo de cabeçalho na linha 21 da Figura 10.21.



Erro comum de programação 10.10

É um erro de compilação incluir a palavra-chave `static` na definição de membros de dados `static` no escopo de arquivo.

Na Figura 10.22, note o uso do operador `new` (linhas 27 e 30) no construtor `Employee` para alocar dinamicamente a quantidade correta de memória para os membros `firstName` e `lastName`. Se o operador `new` for incapaz de atender ao pedido de memória para um ou para esses dois arrays de caracteres, o programa terminará imediatamente. No Capítulo 16, forneceremos um mecanismo melhor para lidar com casos em que `new` não consegue alocar memória.

Note também na Figura 10.22 que as implementações das funções `getFirstName` (linhas 52–58) e `getLastName` (linhas 61–67) retornam ponteiros para os dados de caractere `const`. Nessa implementação, se o cliente quiser reter uma cópia do nome ou do sobrenome, ele será responsável por copiar a memória alocada dinamicamente no objeto `Employee` depois de obter o ponteiro para os dados de caractere `const` do objeto. Também é possível implementar `getFirstName` e `getLastName`, portanto o cliente é solicitado a passar um

array de caracteres e o tamanho dele para cada função. Então as funções poderiam copiar o nome ou o sobrenome no array de caracteres fornecido pelo cliente. Mais uma vez, observe que poderíamos ter utilizado a classe string aqui para retornar uma cópia de um objeto string para o chamador em vez de retornar um ponteiro para os dados private.

```

1 // Figura 10.21: Employee.h
2 // Definição da classe Employee.
3 #ifndef EMPLOYEE_H
4 #define EMPLOYEE_H
5
6 class Employee
7 {
8 public:
9     Employee( const char * const, const char * const ); // construtor
10    ~Employee(); // destrutor
11    const char *getFirstName() const; // retorna o nome
12    const char *getLastName() const; // retorna o sobrenome
13
14    // função-membro static
15    static int getCount(); // retorna número de objetos instanciados
16 private:
17     char *firstName;
18     char *lastName;
19
20     // dados static
21     static int count; // número de objetos instanciados
22 }; // fim da classe Employee
23
24 #endif

```

Figura 10.21 Definição da classe Employee com um membro de dados static para monitorar o número de objetos Employee na memória.

```

1 // Figura 10.22: Employee.cpp
2 // Definições de função-membro da classe Employee.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // protótipos de strlen e strcpy
8 using std::strlen;
9 using std::strcpy;
10
11 #include "Employee.h" // definição da classe Employee
12
13 // define e inicializa o membro de dados static no escopo de arquivo
14 int Employee::count = 0;
15
16 // define a função-membro static que retorna o número de
17 // objetos Employee instanciados (static declarado em Employee.h)
18 int Employee::getCount()
19 {
20     return count;
21 } // fim da função static getCount
22

```

Figura 10.22 Definições de função-membro da classe Employee.

(continua)

```

23 // o construtor aloca dinamicamente espaço para o nome e o sobrenome e
24 // usa strcpy para copiar o nome e o sobrenome para o objeto
25 Employee::Employee( const char * const first, const char * const last )
26 {
27     firstName = new char[ strlen( first ) + 1 ];
28     strcpy( firstName, first );
29
30     lastName = new char[ strlen( last ) + 1 ];
31     strcpy( lastName, last );
32
33     count++; // incrementa contagem estática de empregados
34
35     cout << "Employee constructor for " << firstName
36         << ' ' << lastName << " called." << endl;
37 } // fim do construtor Employee
38
39 // o destrutor desaloca memória dinamicamente alocada
40 Employee::~Employee()
41 {
42     cout << "~Employee() called for " << firstName
43         << ' ' << lastName << endl;
44
45     delete [] firstName; // libera memória
46     delete [] lastName; // libera memória
47
48     count--; // decrementa contagem estática de empregados
49 } // fim do destrutor ~Employee
50
51 // retorna o nome do empregado
52 const char *Employee::getFirstName() const
53 {
54     // const antes do tipo de retorno impede que o cliente modifique
55     // dados private; o cliente deve copiar a string retornada antes de
56     // o destrutor excluir o armazenamento para impedir um ponteiro indefinido
57     return firstName;
58 } // fim da função getFirstName
59
60 // retorna sobrenome do empregado
61 const char *Employee::getLastName() const
62 {
63     // const antes do tipo de retorno impede que o cliente modifique
64     // dados private; o cliente deve copiar a string retornada antes de
65     // o destrutor excluir o armazenamento para impedir um ponteiro indefinido
66     return lastName;
67 } // fim da função getLastName

```

Figura 10.22 Definições de função-membro da classe Employee.

(continuação)

A Figura 10.23 utiliza a função-membro `static` `getCount` para determinar o número de objetos `Employee` atualmente instanciados. Observe que, quando nenhum objeto é instanciado no programa, a chamada de função `Employee::getCount()` é emitida (linhas 14 e 38). Entretanto, quando objetos são instanciados, a função `getCount` pode ser chamada por qualquer um dos objetos, como mostrado na instrução das linhas 22–23 que utiliza o ponteiro `e1Ptr` para invocar a função `getCount`. Observe que utilizar `e2Ptr->getCount()` ou `Employee::getCount()` na linha 23 produziria o mesmo resultado, porque `getCount` sempre acessa o mesmo membro `static` `count`.



Observação de engenharia de software 10.11

Algumas organizações especificam em seus padrões de engenharia de software que todas as chamadas a funções-membro `static` são feitas utilizando o nome da classe, e não um handle de objeto.

```

1 // Figura 10.23: fig10_23.cpp
2 // Driver para testar a classe Employee.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Employee.h" // Definição da classe Employee
8
9 int main()
10 {
11     // utiliza o nome da classe e o operador de resolução de escopo binário para
12     // acessar a função static number getCount
13     cout << "Number of employees before instantiation of any objects is "
14         << Employee::getCount() << endl; // utiliza o nome da classe
15
16     // utiliza new para criar dinamicamente dois novos Employees
17     // operador new também chama o construtor do objeto
18     Employee *e1Ptr = new Employee( "Susan", "Baker" );
19     Employee *e2Ptr = new Employee( "Robert", "Jones" );
20
21     // chama getCount no primeiro objeto Employee
22     cout << "Number of employees after objects are instantiated is "
23         << e1Ptr->getCount();
24
25     cout << "\n\nEmployee 1: "
26         << e1Ptr->getFirstName() << " " << e1Ptr->getLastName()
27         << "\nEmployee 2: "
28         << e2Ptr->getFirstName() << " " << e2Ptr->getLastName() << "\n\n";
29
30     delete e1Ptr; // desaloca memória
31     e1Ptr = 0; // desconecta o ponteiro do espaço de armazenamento livre
32     delete e2Ptr; // desaloca memória
33     e2Ptr = 0; // desconecta o ponteiro do espaço de armazenamento livre
34
35     // não existe nenhum objeto, portanto chama a função-membro static getCount
36     // utilizando o nome da classe e o operador de resolução de escopo binário
37     cout << "Number of employees after objects are deleted is "
38         << Employee::getCount() << endl;
39     return 0;
40 } // fim de main

```

```

Number of employees before instantiation of any objects is 0
Employee constructor for Susan Baker called.
Employee constructor for Robert Jones called.
Number of employees after objects are instantiated is 2

```

```

Employee 1: Susan Baker
Employee 2: Robert Jones

```

```

~Employee() called for Susan Baker
~Employee() called for Robert Jones
Number of employees after objects are deleted is 0

```

Figura 10.23 Membro de dados static monitorando o número de objetos de uma classe.

Uma função-membro deve ser declarada `static` se não acessar membros de dados não-`static` ou funções-membro não-`static` da classe. Ao contrário das funções-membro não-`static`, a função-membro `static` não tem um ponteiro `this`, porque os membros de dados `static` e as funções-membro `static` existem independentemente de qualquer objeto de uma classe. O ponteiro `this` deve referenciar um objeto específico da classe e, quando uma função-membro `static` for chamada, talvez não haja nenhum objeto de sua classe na memória.



Erro comum de programação 10.11

Utilizar o ponteiro `this` em uma função-membro `static` é um erro de compilação.



Erro comum de programação 10.12

Declarar uma função-membro `static const` é um erro de compilação. O qualificador `const` indica que uma função não pode modificar o conteúdo do objeto em que ela opera, mas as funções-membro `static` existem e operam independentemente de qualquer objeto da classe.

As linhas 18–19 da Figura 10.23 utilizam o operador `new` para alocar dinamicamente dois objetos `Employee`. Lembre-se de que o programa terminará imediatamente se não conseguir alocar um ou ambos os objetos. Quando cada objeto `Employee` é alocado, seu construtor é chamado. Quando `delete` é utilizado nas linhas 30 e 32 para desalocar os dois objetos `Employee`, o destrutor de cada objeto é chamado.



Dica de prevenção de erro 10.2

Depois de excluir a memória alocada dinamicamente, configure como 0 o ponteiro que referenciava essa memória. Isso desconecta o ponteiro do espaço anteriormente alocado no armazenamento livre. Esse espaço na memória ainda poderia conter informações, apesar de ter sido excluído. Configurando o ponteiro como 0, o programa perde qualquer acesso a esse espaço de armazenamento livre, o qual, de fato, já poderia ter sido realocado para um propósito diferente. Se você não configurasse o ponteiro como 0, o código poderia acessar inadvertidamente essas novas informações, causando erros de lógica extremamente sutis e não repetíveis.

10.8 Abstração de dados e ocultamento de informações

Em geral, uma classe oculta de seus clientes os detalhes da sua implementação. Isso se chama ocultamento de informações. Como um exemplo do ocultamento de informações, consideremos a estrutura de dados de pilha introduzida na Seção 6.11.

As pilhas podem ser implementadas com arrays e outras estruturas de dados, como listas vinculadas. (Discutimos as pilhas e listas vinculadas no Capítulo 14, “Templates”, e no Capítulo 21, “Estruturas de dados”.) Um cliente de uma classe na pilha não precisa se preocupar com a implementação da pilha. O cliente sabe apenas que, quando itens de dados forem colocados na pilha, eles serão chamados novamente na ordem do último a entrar, primeiro a sair. O cliente se preocupa com *qual* funcionalidade a pilha oferece, não com *como* essa funcionalidade é implementada. Esse conceito é referido como **abstração de dados**. Embora os programadores talvez conheçam os detalhes da implementação de uma classe, eles não devem escrever código que dependa desses detalhes. Isso permite a uma classe particular (como uma que implementa uma pilha e suas operações, *inserir* e *remover*) ser substituída por uma outra versão sem afetar o restante do sistema. Contanto que os serviços públicos da classe não mudem (isto é, cada função-membro público original ainda tem o mesmo protótipo na nova definição de classe), o restante do sistema não é afetado.

Muitas linguagens de programação enfatizam as ações. Nessas linguagens, os dados existem para suportar as ações que os programas devem tomar. Os dados são ‘menos interessantes’ que as ações. Os dados são ‘brutos’. Há apenas alguns tipos de dados predefinidos e é difícil para os programadores criarem seus próprios tipos. O C++ e o estilo de programação orientado a objetos elevam a importância dos dados. As principais atividades da programação em C++ orientada a objetos são a criação de tipos (isto é, classes) e a expressão das interações entre os objetos desses tipos. Para criar linguagens que enfatizam dados, a comunidade das linguagens de programação precisou formalizar algumas noções sobre os dados. A formalização que iremos considerar aqui é a noção de **tipos de dados abstratos** (*abstract data types – ADTs*), que melhora o processo de desenvolvimento de programas.

O que é um tipo de dado abstrato? Considere o tipo predefinido `int`, que a maioria das pessoas associaria com um inteiro em matemática. Em vez disso, um `int` é uma representação abstrata de um inteiro. Diferentemente dos inteiros matemáticos, `ints` de computador têm tamanho fixo. Por exemplo, o tipo `int` nas máquinas populares de 32 bits atuais é, em geral, limitado ao intervalo de $-2.147.483.648$ a $+2.147.483.647$. Se o resultado de um cálculo cai fora desse intervalo, ocorre um erro de estouro e o computador responde de alguma maneira dependente da máquina. Por exemplo, ele poderia produzir um resultado incorreto ‘silenciosamente’, como um valor muito grande para caber em uma variável `int` (comumente chamado **estouro aritmético**). Inteiros matemáticos não têm esse problema. Portanto, a noção de um `int` de computador é somente uma aproximação da noção de um inteiro do mundo real. O mesmo se aplica a um `double`.

Mesmo `char` é uma aproximação; os valores `char` são normalmente padrões de oito bits de uns e zeros; esses padrões não se parecem em nada com os caracteres que representam, como uma letra Z maiúscula, uma letra z minúscula, um sinal de cifrão (\$), um dígito (5) e assim por diante. Os valores do tipo `char` na maioria dos computadores são limitados comparados ao intervalo de caracteres do mundo

real. O conjunto de caracteres ASCII de sete bits (Apêndice B) oferece 128 valores de caracteres diferentes. Isso é inadequado para representar línguas como japonês e chinês, que requerem milhares de caracteres. Como o uso da Internet e da World Wide Web torna-se cada vez mais disseminado, a popularidade do mais novo conjunto de caracteres Unicode está crescendo rapidamente, devido à sua capacidade de representar os caracteres da maioria das línguas. Para informações adicionais sobre o Unicode, visite www.unicode.org.

A questão é que mesmo os tipos de dados predefinidos fornecidos pelas linguagens de programação como o C++ são realmente apenas aproximações ou modelos imperfeitos de conceitos e comportamentos do mundo real. Até aqui, assumimos `int` como algo garantido, mas agora temos uma nova perspectiva a considerar. Tipos como `int`, `double`, `char` e outros são todos exemplos de tipos de dados abstratos. Eles são essencialmente maneiras de representar noções do mundo real em algum nível satisfatório de precisão dentro de um sistema de computador.

Um tipo de dados abstrato na realidade captura duas noções: uma **representação de dados** e as **operações** que podem ser realizadas nesses dados. Por exemplo, em C++, um `int` contém um valor do tipo inteiro (dados) e fornece as operações adição, subtração, multiplicação, divisão e módulo (entre outras) — a divisão por zero é indefinida. Essas operações permitidas são realizadas de uma maneira sensível a parâmetros de máquina, como o tamanho fixo de palavras do sistema de computador subjacente. Outro exemplo é a noção de inteiros negativos cujas operações e representação dos dados são claras, mas a operação de aceitar a raiz quadrada de um inteiro negativo é indefinida. Em C++, o programador utiliza classes para implementar os tipos de dados abstratos e seus serviços. Por exemplo, para implementar um ADT de pilha, criamos nossas próprias classes de pilha no Capítulo 14, “Templates”, e no Capítulo 21, “Estruturas de dados”, e estudamos a biblioteca-padrão da classe `stack` no Capítulo 23, “Standard Template Library (STL)”.

10.8.1 Exemplo: tipo de dados abstrato array

Discutimos os arrays no Capítulo 7. Como descrito, um array não é nada mais do que um ponteiro e um espaço na memória. Essa capacidade primitiva é aceitável para realizar operações de array se o programador for atencioso e flexível. Há muitas operações que seriam elegantes se realizadas com arrays, mas que não são construídas em C++. Com as classes C++, o programador pode desenvolver um ADT de array que é preferível aos arrays ‘brutos’. A classe de array pode fornecer muitas novas capacidades úteis como

- verificação de intervalo de subscrito;
- um intervalo arbitrário de subscritos em vez de ter de iniciar com 0;
- atribuição de array;
- comparação de array;
- entrada/saída de array;
- arrays que sabem seus tamanhos;
- arrays que se expandem dinamicamente para acomodar mais elementos;
- arrays que podem imprimir a si mesmos em um formato tabular organizado.

Criamos nossas próprias classes de array com muitas dessas capacidades no Capítulo 11, “Sobrecarga dos operadores; objetos `string` e array”. Lembre-se de que o template da classe `vector` da C++ Standard Library (introduzido no Capítulo 7) também fornece muitas dessas capacidades. O Capítulo 23 explica o template da classe `vector` em detalhes. O C++ tem um pequeno conjunto de tipos predefinidos. As classes estendem a linguagem de programação básica com novos tipos.



Observação de engenharia de software 10.12

O programador é capaz de criar novos tipos pelo mecanismo de classe. Esses novos tipos podem ser projetados para ser utilizados tão convenientemente quanto os tipos predefinidos. Portanto, o C++ é uma linguagem extensível. Embora a linguagem seja fácil de estender com esses novos tipos, a linguagem básica em si não pode ser alterada.

Novas classes criadas em ambientes C++ podem ser ‘proprietárias’ (isto é, patenteadas) de um indivíduo, pequenos grupos ou empresas. As classes também podem ser colocadas em bibliotecas de classes padrão destinadas à ampla distribuição. O ANSI (American National Standards Institute) e a ISO (International Organization for Standardization) desenvolveram uma versão-padrão de C++ que inclui uma biblioteca de classes padrão. O leitor que aprende C++ e a programação orientada a objetos estará pronto para tirar proveito dos novos tipos de desenvolvimento rápido de softwares orientado a componentes possibilitados por bibliotecas cada vez mais abundantes e ricas.

10.8.2 Exemplo: tipo de dados abstrato string

O C++ é uma linguagem intencionalmente esparsa que fornece aos programadores somente as capacidades elementares necessárias para criar uma ampla série de sistemas (considere-o uma ferramenta para fazer ferramentas). A linguagem é projetada para minimizar o fardo dos problemas de desempenho. O C++ é uma linguagem apropriada para a programação tanto de aplicativos como de sistemas — esta última impõe uma extraordinária demanda de desempenho sobre os programas. Certamente, teria sido possível incluir um tipo de dados `string` entre os tipos de dados predefinidos do C++. Em vez disso, a linguagem foi projetada incluindo mecanismos para criar

e implementar tipos de dados abstratos `string` por meio de classes. Introduzimos a classe `string` da C++ Standard Library no Capítulo 3 e desenvolveremos nossa própria `String` ADT no Capítulo 11. Discutimos a classe `string` em detalhes no Capítulo 18.

10.8.3 Exemplo: tipo de dados abstrato `queue`

Todo mundo uma vez ou outra precisa pegar uma fila. Em inglês uma fila também é chamada de *queue*. Esperamos na fila do caixa do supermercado, esperamos na fila do posto de gasolina, esperamos na fila do ônibus, esperamos na fila do pedágio e todos os estudantes conhecem muito bem a fila de espera para fazer matrícula. Os sistemas de computador utilizam muitas filas de espera internamente, então precisamos escrever programas que simulam o que são filas e o que elas fazem.

Uma fila é um bom exemplo de um tipo de dados abstrato. As filas oferecem comportamentos bem conhecidos para seus clientes. Os clientes colocam uma coisa por vez em uma fila — invocando a operação de **enfileiramento** [*enqueue*] da fila — e recuperam essas coisas individualmente por demanda — invocando a operação de **desenfileiramento** [*dequeue*] da fila. Conceitualmente, uma fila pode tornar-se infinitamente longa. Uma fila real é naturalmente finita. Os itens são retornados de uma fila na ordem **primeiro a entrar, primeiro a sair** (*first-in, first-out* – **FIFO**) — o primeiro item inserido na fila é o primeiro item removido dela.

A fila oculta uma representação interna dos dados que de algum modo monitora os itens que atualmente esperam na fila e oferece um conjunto de operações para seus clientes, a saber, o **enfileiramento** e o **desenfileiramento**. Os clientes não estão preocupados com a implementação da fila. Eles simplesmente querem que a fila opere ‘como anunciado’. Quando um cliente enfileira um novo item, a fila deve aceitar esse item e colocá-lo internamente em alguma estrutura de dados do tipo primeiro a entrar, primeiro a sair. Quando o cliente quiser o próximo item na frente da fila, esta deve remover o item de sua representação interna e entregá-lo para o mundo exterior (isto é, o cliente da fila) na ordem FIFO (isto é, o item que esteve na fila por mais tempo deve ser o próximo retornado pela operação de **desenfileiramento** seguinte).

A fila ADT garante a integridade de sua estrutura de dados interna. Os clientes não podem manipular diretamente essa estrutura de dados. Apenas as funções-membro de fila têm acesso aos seus dados internos. Os clientes podem fazer com que somente as operações admissíveis sejam realizadas na representação de dados; operações não fornecidas na interface pública do ADT são rejeitadas de alguma maneira apropriada. Isso poderia significar emitir uma mensagem de erro, lançar uma exceção (ver Capítulo 16), terminar a execução ou simplesmente ignorar a solicitação de operação.

Criamos nossa própria classe de fila no Capítulo 21, “Estruturas de dados”, e estudamos a classe `queue` da Standard Library no Capítulo 23, “Standard Template Library (STL)”.

10.9 Classes contêineres e iteradores

Entre os tipos de classes mais populares estão as **classes contêineres** (também chamadas **classes de coleção**), isto é, classes projetadas para armazenar coleções de objetos. As classes contêineres fornecem comumente serviços como inserção, exclusão, pesquisa, classificação e teste de um item para determinar se ele é ou não um membro da coleção. Os arrays, pilhas, filas, árvores e listas vinculadas são exemplos de classes contêineres; estudamos arrays no Capítulo 7 e estudaremos cada uma dessas outras estruturas de dados no Capítulo 21, “Estruturas de dados”, e no Capítulo 23, “Standard Template Library (STL)”.

É comum associar **objetos iteradores** — ou mais simplesmente **iteradores** — com classes contêineres. O iterador é um objeto que ‘percorre’ uma coleção, retornando o próximo item (ou realizando alguma ação nele). Uma vez que um iterador de uma classe foi escrito, obter o próximo elemento da classe pode ser expressado de maneira simples. Exatamente como um livro compartilhado por várias pessoas poderia ter vários marcadores de uma vez, uma classe contêiner pode ter vários iteradores operacionais de uma vez. Cada iterador mantém suas próprias informações de ‘posição’. Discutiremos contêineres e iteradores em detalhes no Capítulo 23.

10.10 Classes proxy

Lembre-se de que dois dos princípios fundamentais da boa engenharia de software são separar a interface da implementação e ocultar detalhes da implementação. Esforçamo-nos para alcançar esses objetivos definindo uma classe em um arquivo de cabeçalho e implementando suas funções-membro em um arquivo de implementação separado. Entretanto, como indicamos no Capítulo 9, os arquivos de cabeçalho *realmente* contêm alguma parte da implementação de uma classe e dicas sobre outras. Por exemplo, os membros `private` de uma classe são listados na definição de classe em um arquivo de cabeçalho, então esses membros são visíveis aos clientes, mesmo que os clientes não possam acessar os membros `private`. Revelar os dados `private` de uma classe dessa maneira potencialmente expõe informações proprietárias aos clientes da classe. Agora introduzimos a noção de uma **classe proxy** que permite ocultar dos clientes de uma classe até mesmo os dados `private` dessa classe. Fornecer aos clientes de sua classe uma classe proxy que conhece somente a interface `public` para sua classe permite aos clientes utilizar os serviços da sua classe sem dar acesso de cliente aos detalhes de implementação de sua classe.

Implementar uma classe proxy exige vários passos, que demonstramos nas figuras 10.24–10.27. Primeiro, criamos a definição de classe para a classe que contém a implementação proprietária que gostaríamos de ocultar. Nossa classe de exemplo, chamada `Implementation`, é mostrada na Figura 10.24. A classe proxy `Interface` é mostrada nas figuras 10.25–10.26. O programa de teste e a saída de exemplo são mostrados na Figura 10.27.

A classe `Implementation` (Figura 10.24) fornece um único membro de dados `private` chamado `value` (os dados que gostaríamos de ocultar do cliente), um construtor para inicializar `value` e as funções `setValue` e `getValue`.


```

1 // Figura 10.24: Implementation.h
2 // Arquivo de cabeçalho para a classe Implementation
3
4 class Implementation
5 {
6 public:
7     // construtor
8     Implementation( int v )
9         : value( v ) // inicializa value como v
10    {
11        // corpo vazio
12    } // fim do construtor de Implementation
13
14    // configura valor como v
15    void setValue( int v )
16    {
17        value = v; // deve validar v
18    } // fim da função setValue
19
20    // retorna value
21    int getValue() const
22    {
23        return value;
24    } // fim da função getValue
25 private:
26     int value; // dados que gostaríamos de ocultar do cliente
27 }; // fim da classe Implementation

```

Figura 10.24 Definição da classe Implementation.

```

1 // Figura 10.25: Interface.h
2 // Arquivo de cabeçalho da classe Interface
3 // O cliente vê esse código-fonte, mas o código-fonte não revela
4 // o layout de dados da classe Implementation.
5
6 class Implementation; // declaração de classe antecipada requerida pela linha 17
7
8 class Interface
9 {
10 public:
11     Interface( int ); // construtor
12     void setValue( int ); // mesma interface public que
13     int getValue() const; // a classe Implementation tem
14     ~Interface(); // destrutor
15 private:
16     // requer a declaração antecipada anterior (linha 6)
17     Implementation *ptr;
18 }; // fim da classe Interface

```

Figura 10.25 Definição da classe Interface.

Definimos uma classe proxy chamada `Interface` (Figura 10.25) com uma interface `public` idêntica (exceto pelos nomes do construtor e do destrutor) àquela da classe `Implementation`. O único membro `private` da classe proxy é um ponteiro para um objeto da classe `Implementation`. Utilizar um ponteiro dessa maneira permite ocultar do cliente os detalhes de implementação da classe `Implementation`. Note que as únicas menções na classe `Interface` à classe `Implementation` proprietária estão na declaração de ponteiro (linha 17) e na linha 6, uma **declaração de classe antecipada** [*forward class declaration*]. Quando uma definição de classe (como a classe `Interface`) utiliza somente um ponteiro para ou referência a um objeto de outra classe (como um objeto da classe `Implementation`), o arquivo de cabeçalho de classe para essa outra classe (que comumente revelaria os dados `private` dessa classe) não precisa ser incluído com `#include`. Você pode simplesmente declarar essa outra classe como um tipo de dados com uma declaração de classe antecipada (linha 6) antes de o tipo ser utilizado no arquivo.

O arquivo de implementação de função-membro da classe proxy `Interface` (Figura 10.26) é o único arquivo que inclui o arquivo de cabeçalho `Implementation.h` (linha 5) contendo a classe `Implementation`. O arquivo `Interface.cpp` (Figura 10.26) é fornecido para o cliente como um arquivo de código-objeto pré-compilado junto com o arquivo de cabeçalho `Interface.h` que inclui os protótipos de função dos serviços fornecidos pela classe proxy. Como o arquivo `Interface.cpp` é disponibilizado para o cliente somente como código-objeto, o cliente não é capaz de ver as interações entre a classe proxy e a classe proprietária (patenteada) (linhas 9, 17, 23 e 29). Note que a classe proxy impõe uma ‘camada’ extra de chamadas de função como o ‘preço a pagar’ para ocultar dados `private` de implementação da classe. Considerando a velocidade dos computadores atuais e o fato de que muitos compiladores podem colocar `inline` chamadas de função simples automaticamente, o efeito dessas chamadas de função extra no desempenho é frequentemente insignificante.

A Figura 10.27 testa a classe `Interface`. Note que apenas o arquivo de cabeçalho de `Interface` é incluído no código-cliente (linha 7) — não há nenhuma menção da existência de uma classe separada chamada `Implementation`. Portanto, o cliente nunca vê os dados `private` da classe `Implementation`, nem o código-cliente pode tornar-se dependente do código `Implementation`.



Observação de engenharia de software 10.13

Uma classe proxy isola o código-cliente das alterações na implementação.

```

1  // Figura 10.26: Interface.cpp
2  // Implementação da classe Interface - o cliente recebe somente esse arquivo
3  // como código-objeto pré-compilado, mantendo a implementação oculta.
4  #include "Interface.h" // Definição da classe Interface
5  #include "Implementation.h" // Definição da classe Implementation
6
7  // construtor
8  Interface::Interface( int v )
9      : ptr ( new Implementation( v ) ) // inicializa ptr para apontar para
10 {                                     // um novo objeto Implementation
11     // corpo vazio
12 } // fim do construtor Interface
13
14 // chama a função setValue de Implementation
15 void Interface::setValue( int v )
16 {
17     ptr->setValue( v );
18 } // fim da função setValue
19
20 // chama a função getValue de Implementation
21 int Interface::getValue() const
22 {
23     return ptr->getValue();
24 } // fim da função getValue
25
26 // destrutor
27 Interface::~Interface()
28 {
29     delete ptr;
30 } // fim do destrutor ~Interface

```

Figura 10.26 Definições de função-membro da classe `Interface`.

```

1 // Figura 10.27: fig10_27.cpp
2 // Ocultando dados private de uma classe com a classe proxy.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Interface.h" // Definição da classe Interface
8
9 int main()
10 {
11     Interface i( 5 ); // cria objeto Interface
12
13     cout << "Interface contains: " << i.getValue()
14         << " before setValue" << endl;
15
16     i.setValue( 10 );
17
18     cout << "Interface contains: " << i.getValue()
19         << " after setValue" << endl;
20     return 0;
21 } // fim de main

```

```

Interface contains: 5 before setValue
Interface contains: 10 after setValue

```

Figura 10.27 Implementando uma classe proxy.

10.11 Síntese

Neste capítulo, introduzimos vários tópicos avançados relacionados com classes e abstração de dados. Você aprendeu a especificar objetos `const` e funções-membro `const` para impedir modificações em objetos, impondo assim o princípio do menor privilégio. Você também aprendeu que, por composição, uma classe pode ter objetos de outras classes como membros. Introduzimos o tópico de amizade [*friendship*] e apresentamos exemplos que demonstram como utilizar funções `friend`.

Você aprendeu que o ponteiro `this` é passado como um argumento implícito para cada uma das funções-membro não-`static` de uma classe, permitindo às funções acessar membros de dados do objeto correto e outras funções-membro não-`static`. Você também viu o uso explícito do ponteiro `this` para acessar os membros da classe e permitir chamadas de função-membro em cascata.

O capítulo introduziu o conceito de gerenciamento de memória dinâmico. Você aprendeu que programadores em C++ podem criar e destruir objetos dinamicamente com os operadores `new` e `delete`. Motivamos a necessidade dos membros de dados `static` e demonstramos como declarar e utilizar membros de dados `static` e funções-membro em suas próprias classes.

Você aprendeu a abstração de dados e o ocultamento de informações — dois dos conceitos fundamentais da programação orientada a objetos. Discutimos os tipos de dados abstratos — as maneiras de representar noções do mundo real ou noções conceituais em um nível satisfatório de precisão dentro de um sistema de computador. Então você aprendeu sobre os três tipos de dados abstratos de exemplo — arrays, strings e filas. Introduzimos o conceito de uma classe contêiner que armazena uma coleção de objetos, bem como a noção de uma classe iteradora que percorre os elementos de uma classe contêiner. Por fim, você aprendeu a criar uma classe proxy para ocultar os detalhes de implementação (incluindo os membros de dados `private`) de uma classe dos clientes da classe.

No Capítulo 11, continuamos nosso estudo de classes e objetos mostrando como permitir que operadores do C++ funcionem com objetos — um processo chamado sobrecarga dos operadores. Por exemplo, você verá como ‘sobrecarregar’ o operador `<<` de modo que ele possa ser utilizado para gerar saída de um array completo sem usar uma instrução de repetição explicitamente.

Resumo

- A palavra-chave `const` pode ser utilizada para especificar que um objeto não é modificável e que qualquer tentativa de modificá-lo deve resultar em um erro de compilação.
- Os compiladores C++ não permitem chamadas de função-membro não-`const` em objetos `const`.
- Uma tentativa de uma função-membro `const` modificar um objeto de sua classe (`*this`) é um erro de compilação.

- Uma função é especificada como `const` tanto em seu protótipo como em sua definição.
- Um objeto `const` deve ser inicializado, não atribuído.
- Os construtores e destrutores não podem ser declarados `const`.
- O membro de dados `const` e os membros de dados que são referências devem ser inicializados com inicializadores de membro.
- Uma classe pode ter objetos de outras classes como membros — esse conceito é chamado composição.
- Os objetos-membro são construídos na ordem em que são declarados na definição de classe e antes de seus objetos da classe contêiner serem construídos.
- Se um objeto-membro não receber um inicializador de membro, o construtor-padrão do objeto-membro será chamado implicitamente.
- Uma função `friend` de uma classe é definida fora do escopo dessa classe, ainda que ela tenha o direito de acessar membros não-`public` e `public` da classe. Funções independentes ou classes inteiras podem ser declaradas amigas de outra classe.
- Uma declaração amiga pode aparecer em qualquer lugar da classe. Uma amiga é essencialmente uma parte da interface `public` da classe.
- A relação de amizade não é simétrica nem transitiva.
- Cada objeto tem acesso ao seu próprio endereço pelo ponteiro `this`.
- O ponteiro `this` de um objeto não faz parte do objeto em si — isto é, o tamanho da memória ocupado pelo ponteiro `this` não é refletido no resultado de uma operação `sizeof` no objeto.
- O ponteiro `this` é passado (pelo compilador) como um argumento implícito para cada uma das funções não-`static` membro do objeto.
- Os objetos utilizam o ponteiro `this` implicitamente (como fizemos até agora) ou explicitamente para referenciar seus membros de dados e funções-membro.
- O ponteiro `this` permite chamadas de função-membro em cascata em que múltiplas funções são invocadas na mesma instrução.
- O gerenciamento de memória dinâmico permite aos programadores controlar a alocação e a desalocação de memória em um programa para qualquer tipo predefinido ou definido pelo usuário.
- O armazenamento livre (às vezes chamado de `heap`) é uma região da memória atribuída a cada programa para armazenar objetos dinamicamente alocados em tempo de execução.
- O operador `new` aloca o armazenamento do tamanho adequado para um objeto, executa o construtor do objeto e retorna um ponteiro do tipo correto. O operador `new` pode ser utilizado para alocar dinamicamente qualquer tipo fundamental (como `int` ou `double`) ou tipo de classe. Se `new` não conseguir localizar o espaço na memória para o objeto, ele indica que ocorreu um erro ‘lançando’ uma ‘exceção’. Isso normalmente faz com que o programa termine imediatamente.
- Para destruir um objeto dinamicamente alocado e liberar o espaço para o objeto, utilize o operador `delete`.
- Um array de objetos pode ser alocado dinamicamente com `new` como em

```
int *ptr = new int[ 100 ];
```

que aloca um array de 100 inteiros e atribui a localização inicial do array a `ptr`. O array de inteiros anterior é excluído com a instrução

```
delete [] ptr;
```

- Um membro de dados `static` representa informações ‘no nível de classe’ (isto é, uma propriedade da classe compartilhada por todas as instâncias, não uma propriedade de um objeto específico da classe).
- Membros de dados `static` têm escopo de classe e podem ser declarados `public`, `private` ou `protected`.
- Os membros `static` de uma classe existem até mesmo quando não existe nenhum objeto dessa classe.
- Para acessar um membro de classe `public static` quando não existe nenhum objeto da classe, simplesmente prefixe o nome de classe e o operador de resolução de escopo binário (`::`) com o nome do membro de dados.
- Os membros de classe `public static` de uma classe podem ser acessados por qualquer objeto dessa classe.
- Uma função-membro deve ser declarada `static` se não acessar membros de dados não-`static` ou funções-membro não-`static` da classe. Ao contrário das funções-membro não-`static`, a função-membro `static` não tem um ponteiro `this`, porque os membros de dados `static` e as funções-membro `static` existem independentemente de qualquer objeto de uma classe.
- Os tipos de dados abstratos são formas de representar noções do mundo real e noções conceituais em um nível satisfatório de precisão dentro de um sistema de computador.
- Um tipo de dados abstrato captura duas noções: uma representação de dados e as operações que podem ser realizadas nesses dados.
- O C++ é uma linguagem intencionalmente esparsa que fornece aos programadores as capacidades elementares necessárias para criar uma ampla série de sistemas. O C++ é projetado para minimizar cargas de desempenho.
- Os itens são retornados de uma fila na ordem primeiro a entrar, primeiro a sair (*first-in, first-out* – FIFO) — o primeiro item inserido na fila é o primeiro item removido dela.

- As classes contêineres (também chamadas classes de coleção) são projetadas para armazenar coleções de objetos. As classes contêineres fornecem comumente serviços como inserção, exclusão, pesquisa, classificação e teste de um item para determinar se ele é ou não um membro da coleção.
- É comum associar iteradores com classes contêineres. O iterador é um objeto que ‘percorre’ uma coleção, retornando o próximo item (ou realizando alguma ação nele).
- Fornecer aos clientes de sua classe uma classe proxy que conhece apenas a interface `public` para a sua classe permite que eles utilizem os serviços da sua classe sem ter acesso aos detalhes de implementação, como seus dados `private`.
- Quando uma definição de classe utiliza apenas um ponteiro ou referência para um objeto de outra classe, o arquivo de cabeçalho de classe para essa outra classe (que comumente revelaria os dados `private` dessa classe) não precisa ser incluído com `#include`. Você pode simplesmente declarar essa outra classe como um tipo de dados com uma declaração de classe antecipada antes de o tipo ser utilizado no arquivo.
- O arquivo de implementação que contém funções-membro de uma classe proxy é o único arquivo que inclui o arquivo de cabeçalho para a classe cujos dados `private` gostaríamos de ocultar.
- O arquivo de implementação contendo as funções-membro para a classe proxy é fornecido ao cliente como um arquivo de código-objeto pré-compilado junto com o arquivo de cabeçalho que inclui os protótipos de função dos serviços fornecidos pela classe proxy.

Terminologia

abstração de dados	desalocar memória	objeto host
alocar memória	desenfileiramento (operação de fila)	objetos dinâmicos
armazenamento livre	enfileiramento (operação de fila)	ocultamento de informações
chamadas de função-membro em cascata	enfileiramento de tipo de dados abstrato	operações em um ADT
classe contêiner	estouro aritmético	primeiro a entrar, primeiro a sair (<i>first-in, first-out</i> – FIFO)
classe de coleção	friend, classe	representação de dados
classe proxy	friend, função	static, função-membro
composição	gerenciamento de memória dinâmico	static, membro de dados
const, função-membro	heap	tem um, relacionamento
const, objeto	inicializador de membro	this, ponteiro
construtor de objeto-membro	iterador	tipo de dados abstrato (ADT)
controle de acesso de membro	lista de inicializadores de membro	último a entrar, primeiro a sair (<i>last-in, first-out</i> – LIFO)
declaração de classe antecipada	memória, vazamento	
delete, operador	new, operador	
delete[], operador	new[], operador	

Exercícios de revisão

- 10.1** Preencha as lacunas em cada uma das seguintes sentenças:
- Os (As) _____ devem ser utilizados(as) para inicializar membros constantes de uma classe.
 - Uma função não-membro deve ser declarada como _____ de uma classe para ter acesso aos membros de dados `private` dessa classe.
 - O operador _____ aloca memória dinamicamente para um objeto de um tipo especificado e retorna um _____ para esse tipo.
 - Um objeto constante deve ser _____; depois de criado, ele não pode ser modificado.
 - Um membro de dados _____ representa informações no nível da classe.
 - As funções-membro não-`static` de um objeto têm acesso a um ‘autoponteiro’ para o objeto chamado ponteiro _____.
 - A palavra-chave _____ especifica que um objeto ou variável não é modificável depois de inicializado.
 - Se um inicializador de membro não receber um objeto-membro de uma classe, o objeto _____ é chamado.
 - Uma função-membro deve ser declarada `static` se não acessar os membros de classe _____.
 - Os objetos-membro são construídos _____ do objeto da classe contêiner.
 - O operador _____ reivindica memória anteriormente alocada por `new`.
- 10.2** Localize os erros na seguinte classe e explique como corrigi-los:

```
class Example
{
public:
    Example( int y = 10 )
        : data( y )
    {
```

```

        // corpo vazio
    } // fim do construtor Example

    int getIncrementedData() const
    {
        return data++;
    } // fim da função getIncrementedData
    static int getCount()
    {
        cout << "Data is " << data << endl;
        return count;
    } // fim da função getCount
private:
    int data;
    static int count;
}; // fim da classe Example

```

Respostas dos exercícios de revisão

- 10.1** a) inicializadores de membro. b) friend. c) new, ponteiro. d) inicializado. e) static. f) this. g) const. h) construtor-padrão. i) não-static. j) antes. k) delete.
- 10.2** Erro: A definição de classe para Example tem dois erros. O primeiro ocorre na função getIncrementedData. A função é declarada const, mas modifica o objeto.
Correção: Para corrigir o primeiro erro, remova a palavra-chave const da definição de getIncrementedData.
Erro: O segundo erro ocorre na função getCount. Essa função é declarada static, portanto não tem permissão de acessar nenhum membro da classe não-static.
Correção: Para corrigir o segundo erro, remova a linha de saída da definição getCount.

Exercícios

- 10.3** Compare e contraste os operadores de alocação e de desalocação dinâmica de memória new, new [], delete e delete [].
- 10.4** Explique a noção de amizade em C++. Explique os aspectos negativos de amizade como descritos no texto.
- 10.5** Uma definição da classe Time correta pode incluir os dois construtores a seguir? Se não, explique por quê.
- ```

Time(int h = 0, int m = 0, int s = 0);
Time();

```
- 10.6** O que acontece quando um tipo de retorno, mesmo void, é especificado para um construtor ou destrutor?
- 10.7** Modifique a classe Date na Figura 10.10 para obter as seguintes capacidades:
- a) Dar saída para a data em múltiplos formatos como
- ```

DDD YYYY
MM/DD/YY
June 14, 1992

```
- b) Utilizar construtores sobrecarregados para criar objetos Date inicializados com datas dos formatos na parte (a).
- c) Criar um construtor Date que lê a data de sistema utilizando as funções-padrão de biblioteca do cabeçalho <ctime> e configurar os membros Date. (Consulte a documentação de referência do seu compilador ou www.cplusplus.com/ref/ctime/index.html para obter informações sobre as funções no cabeçalho <ctime>.)
- No Capítulo 11, seremos capazes de criar operadores para testar a igualdade de duas datas e compará-las para determinar se uma vem antes ou depois da outra.
- 10.8** Crie uma classe SavingsAccount. Utilize um membro de dados static annualInterestRate para armazenar a taxa de juros anual para cada um dos correntistas. Cada membro da classe contém um membro de dados private savingsBalance para indicar a quantia que os correntistas têm atualmente em depósito. Forneça a função-membro calculateMonthlyInterest que calcula os juros mensais multiplicando o balance [saldo] pelo annualInterestRate dividido por 12; esses juros devem ser adicionados a savingsBalance. Forneça uma função-membro static modifyInterestRate que configura o static annualInterestRate com um novo valor. Escreva um programa de driver para testar a classe SavingsAccount. Instancie dois objetos diferentes da classe SavingsAccount, saver1 e saver2, com saldos de \$ 2.000,00 e \$ 3.000,00, respectivamente. Configure o annualInterestRate como 3%. Em seguida, calcule os juros mensais e imprima os novos saldos de cada um dos correntistas. Então configure o annualInterestRate como 4%, calcule os juros do próximo mês e imprima os novos saldos para cada um dos poupadores.

10.9 Crie a classe `IntegerSet` pela qual cada objeto pode armazenar inteiros no intervalo 0 a 100. Um conjunto é representado internamente como um array de uns e zeros. O elemento do array `a[ i ]` é 1 se o inteiro i estiver no conjunto. O elemento do array `a[ j ]` é `false` se o inteiro j não estiver no conjunto. O construtor-padrão inicializa um conjunto para o chamado ‘conjunto vazio’, isto é, um conjunto cuja representação de array só contém zeros.

Forneça funções-membro para as operações comuns de conjuntos. Por exemplo, forneça uma função-membro `unionOfSets` que cria um terceiro conjunto que seja a união teórica de dois conjuntos existentes (isto é, um elemento do array do terceiro conjunto é configurado como 1 se esse elemento for 1 em qualquer um dos conjuntos existentes, ou em ambos, e um elemento do array do terceiro conjunto é configurado como 0 se esse elemento for 0 em cada um dos conjuntos existentes).

Forneça uma função-membro `intersectionOfSets` que cria um terceiro conjunto que seja a intersecção teórica de dois conjuntos existentes (isto é, um elemento do array do terceiro conjunto é configurado como 0 se esse elemento for 0 em qualquer um ou ambos os conjuntos existentes, e um elemento do array do terceiro conjunto é configurado como 1 se esse elemento for 1 em cada um dos conjuntos existentes).

Forneça uma função-membro `insertElement` que insere um novo inteiro k em um conjunto (configurando `a[ k ]` como 1). Forneça uma função-membro `deleteElement` que exclui o inteiro m (configurando `a[ m ]` como 0).

Forneça uma função-membro `printSet` que imprime um conjunto como uma lista de números separados por espaços. Imprima somente aqueles elementos que estiverem presentes no conjunto (isto é, sua posição no array tem um valor de 1). Imprima `---` para um conjunto vazio.

Forneça uma função-membro `isEqualTo` que determina se dois conjuntos são iguais.

Forneça um construtor adicional que recebe um array de inteiros e o tamanho desse array e utiliza o array para inicializar um objeto configurado.

Agora escreva um programa de driver para testar sua classe `IntegerSet`. Instancie diversos objetos `IntegerSet`. Teste se todas as suas funções-membro funcionam adequadamente.

10.10 Seria perfeitamente razoável para a classe `Time` das figuras 10.18–10.19 representar a hora internamente como o número de segundos desde a meia-noite em vez representá-la com os três valores de inteiro `hour`, `minute` e `second`. Os clientes poderiam utilizar os mesmos métodos `public` e obter os mesmos resultados. Modifique a classe `Time` da Figura 10.18 para implementar a hora como o número de segundos desde a meia-noite e mostrar que não há alteração visível na funcionalidade para os clientes da classe. [Nota: Este exercício demonstra com precisão as virtudes do ocultamento da implementação.]



Toda a diferença entre construção e criação é exatamente esta: que uma coisa construída só pode ser amada depois de ser construída; mas uma coisa criada é amada antes de existir.

Gilbert Keith Chesterton

Os dados foram lançados.
Júlio César

Nosso médico realmente nunca iria nos operar a menos que necessário. Ele era assim. Se não precisasse do dinheiro, nem tocaria em você.
Herb Shriner

Sobrecarga de operadores; objetos string e array

OBJETIVOS

Neste capítulo, você aprenderá:

- O que é sobrecarga de operadores e como ela torna os programas mais legíveis e a programação mais conveniente.
- Como redefinir (sobrecarregar) operadores para trabalhar com objetos de classes definidas pelo usuário.
- As diferenças entre sobrecarregar operadores unários e binários.
- Como converter objetos de uma classe em outra classe.
- Quando sobrecarregar e quando não sobrecarregar operadores.
- Como criar classes `PhoneNumber`, `Array`, `String` e `Date` que demonstram a sobrecarga de operadores.
- Como utilizar operadores sobrecarregados e outras funções-membro da biblioteca de classe-padrão `string`.
- Como utilizar a palavra-chave `explicit` para impedir o compilador de utilizar construtores de um único argumento para realizar conversões implícitas.

- 11.1 Introdução
- 11.2 Fundamentos de sobrecarga de operadores
- 11.3 Restrições à sobrecarga de operadores
- 11.4 Funções operadoras como membros de classe *versus* funções globais
- 11.5 Sobrecarregando operadores de inserção e extração de fluxo
- 11.6 Sobrecarregando operadores unários
- 11.7 Sobrecarregando operadores binários
- 11.8 Estudo de caso: classe Array
- 11.9 Convertendo entre tipos
- 11.10 Estudo de caso: classe String
- 11.11 Sobrecarregando ++ e --
- 11.12 Estudo de caso: uma classe Date
- 11.13 Classe string da biblioteca-padrão
- 11.14 Construtores explicit
- 11.15 Síntese

Resumo | Terminologia | Exercícios de revisão | Respostas dos exercícios de revisão | Exercícios

11.1 Introdução

Os capítulos 9–10 introduziram os princípios básicos de classes em C++. Os serviços eram obtidos de objetos enviando mensagens (na forma de chamadas de função-membro) para os objetos. Essa notação de chamada de função é incômoda para certos tipos de classes (como classes matemáticas). Além disso, muitas manipulações comuns são realizadas com operadores (por exemplo, entrada e saída). Podemos utilizar o rico conjunto de operadores predefinidos do C++ para especificar manipulações comuns de objeto. Este capítulo mostra como permitir que os operadores C++ funcionem com objetos — um processo chamado **sobrecarga de operadores**. É simples e natural estender o C++ com essas novas capacidades, mas isso deve ser feito cautelosamente.

Um exemplo de um operador sobrecarregado construído no C++ é o `<<`, que é utilizado como operador de inserção de fluxo e como operador de bits de deslocamento para a esquerda (que é discutido no Capítulo 22, “Bits, caracteres, strings e structs”). De maneira semelhante, `>>` também é sobrecarregado; é utilizado como operador de extração de fluxo e como operador de bits de deslocamento para a direita. [Nota: Os operadores de bits de deslocamento para a esquerda e para a direita são discutidos em detalhes no Capítulo 22.] Esses dois operadores são sobrecarregados na C++ Standard Library.

Embora a sobrecarga de operadores pareça uma capacidade exótica, a maioria dos programadores utiliza operadores sobrecarregados implícita e regularmente. Por exemplo, a própria linguagem C++ sobrecarrega o operador de adição (+) e o de subtração (-). Esses operadores executam de modo diferente, dependendo do seu contexto na aritmética de inteiros, na aritmética de ponto flutuante e na aritmética de ponteiros.

O C++ permite ao programador sobrecarregar a maioria dos operadores para que eles se tornem sensíveis ao contexto em que são utilizados — o compilador gera o código apropriado com base no contexto (em particular, os tipos dos operandos). Alguns operadores são sobrecarregados frequentemente, em especial o operador de atribuição e vários operadores aritméticos como + e -. Os trabalhos realizados por operadores sobrecarregados também podem ser realizados por chamadas de função explícita, mas a notação de operador é, muitas vezes, mais clara e mais familiar aos programadores.

Discutimos quando utilizar e quando não utilizar sobrecarga de operadores. Implementamos as classes definidas pelo usuário `PhoneNumber`, `Array`, `String` e `Date` para demonstrar como sobrecarregar operadores, incluindo os operadores de inserção de fluxo, de extração de fluxo, de atribuição, de igualdade, de subscrito, relacionais, de negação lógica, parênteses e os operadores de incremento. O capítulo termina com um exemplo da classe `string` da biblioteca-padrão do C++, que fornece muitos operadores sobrecarregados que são semelhantes à classe `String` apresentados anteriormente no capítulo. Nos exercícios, pedimos para você implementar várias classes com os operadores sobrecarregados. Os exercícios também utilizam as classes `Complex` (para números complexos) e `HugeInt` (para inteiros maiores que aqueles que um computador pode representar com o tipo `long`) para demonstrar os operadores aritméticos sobrecarregados + e - e solicitar a você o aprimoramento dessas classes sobrecarregando outros operadores aritméticos.

11.2 Fundamentos de sobrecarga de operadores

A programação em C++ é um processo sensível ao tipo e focalizado no tipo. Os programadores podem utilizar tipos fundamentais e definir novos tipos. Os tipos fundamentais podem ser utilizados com a rica coleção de operadores do C++. Os operadores fornecem aos programadores uma notação concisa para expressar manipulações de objetos de tipos fundamentais.

Os programadores também podem utilizar operadores com tipos definidos pelo usuário. Embora não permita que novos operadores sejam criados, o C++ permite que a maioria dos operadores existentes seja sobrecarregada de modo que, quando forem utilizados com objetos, eles tenham um significado apropriado para esses objetos. Essa é uma capacidade poderosa.



Observação de engenharia de software 11.1

A sobrecarga de operadores contribui para a extensibilidade do C++ — um dos atributos mais atraentes da linguagem.



Boa prática de programação 11.1

Utilize a sobrecarga de operadores quando ela torna um programa mais claro do que realizar as mesmas operações com chamadas de função.



Boa prática de programação 11.2

Os operadores sobrecarregados devem simular a funcionalidade de suas contrapartes predefinidas — por exemplo, o operador + deve ser sobrecarregado para realizar adição, não subtração. Evite o uso excessivo ou inconsistente de sobrecarga de operadores, já que isso pode tornar um programa obscuro e difícil de ler.

Um operador é sobrecarregado escrevendo uma definição de função-membro não-`static` ou uma definição de função global como você escreveria normalmente, exceto pelo fato de que o nome da função agora se torna a palavra-chave `operator` seguida pelo símbolo do operador sendo sobrecarregado. Por exemplo, o nome de função `operator+` seria utilizado para sobrecarregar o operador de adição (+). Quando operadores são sobrecarregados como funções-membro, eles devem ser não-`static`, porque devem ser chamados sobre um objeto da classe e operar sobre esse objeto.

Para utilizar um operador em objetos de classe, esse operador *deve* ser sobrecarregado — com três exceções. O operador de atribuição (=) pode ser utilizado com toda a classe para realizar atribuição de membro a membro dos membros de dados da classe — cada membro de dados é atribuído a partir do objeto ‘origem’ para o objeto ‘alvo’ da atribuição. Logo veremos que essa atribuição de membro a membro padrão é perigosa para classes com membros ponteiro; sobrecarregaremos explicitamente o operador de atribuição para essas classes. Os operadores de endereço (&) e vírgula (,) também podem ser utilizados com objetos de qualquer classe sem a sobrecarga. O operador de endereço retorna o endereço do objeto na memória. O operador vírgula avalia a expressão à sua esquerda e, depois, à sua direita. Esses dois operadores também podem ser sobrecarregados.

Sobrecarregar é especialmente apropriado para classes matemáticas. Essas classes muitas vezes requerem que um conjunto substancial de operadores seja sobrecarregado para assegurar a consistência com a maneira como estas classes matemáticas são tratadas no mundo real. Por exemplo, não seria comum sobrecarregar apenas adição para uma classe de número complexo, porque outros operadores aritméticos também são comumente utilizados com números complexos.

A sobrecarga de operadores fornece as mesmas expressões concisas e familiares para tipos definidos pelo usuário que o C++ fornece com sua rica coleção de operadores para tipos fundamentais. A sobrecarga de operadores não é automática — você deve escrever funções de sobrecarga de operadores para realizar as operações desejadas. Às vezes, essas funções são melhores quando tornadas funções-membro; às vezes são melhores como funções `friend`; ocasionalmente, podem ser funções não-`friend` globais. Discutimos essas questões por todo o capítulo.

11.3 Restrições à sobrecarga de operadores

A maioria dos operadores do C++ pode ser sobrecarregada. Esses operadores são mostrados na Figura 11.1. A Figura 11.2 mostra os operadores que não podem ser sobrecarregados.



Erro comum de programação 11.1

Tentar sobrecarregar um operador não sobrecarregável é um erro de sintaxe.

Precedência, associatividade e número de operandos

A precedência de um operador não pode ser alterada pela sobrecarga. Isso pode levar a situações incômodas em que um operador é sobrecarregado de uma maneira pela qual sua precedência fixa é inadequada. Entretanto, os parênteses podem ser utilizados para forçar a ordem de avaliação de operadores sobrecarregados em uma expressão.

A associatividade de um operador (isto é, se o operador é aplicado da direita para a esquerda ou da esquerda para a direita) não pode ser alterada pela sobrecarga.

Não é possível alterar a ‘aridade’ de um operador (isto é, o número de operandos que um operador aceita): operadores unários sobrecarregados permanecem operadores unários; operadores binários sobrecarregados permanecem operadores binários. O único operador ternário (?:) do C++ não pode ser sobrecarregado. Os operadores &, *, + e - têm tanto a versão unária como a binária; essas versões unária e binária podem ser sobrecarregadas.

Operadores que podem ser sobrecarregados							
+	-	*	/	%	^	&	
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	=	<<	>>	>>=
<<=	==	!=	<=	>=	&&		++
--	->*	,	->	[]	()	new	delete
new[]	delete[]						

Figura 11.1 Operadores que podem ser sobrecarregados.

Operadores que não podem ser sobrecarregados			
.	.*	::	?:

Figura 11.2 Operadores que não podem ser sobrecarregados.



Erro comum de programação 11.2

Tentar alterar a 'aridade' de um operador via sobrecarga de operadores é um erro de compilação.

Criando novos operadores

Não é possível criar novos operadores; apenas os operadores existentes podem ser sobrecarregados. Infelizmente, isso impede o programador de utilizar notações populares como o operador `**` usado em algumas outras linguagens de programação para exponenciação. [Nota: Você poderia sobrecarregar o operador `^` para realizar exponenciação — como ele faz em algumas outras linguagens.]



Erro comum de programação 11.3

Tentar criar novos operadores via sobrecarga de operadores é um erro de sintaxe.

Operadores para tipos fundamentais

O significado de como um operador funciona em objetos de tipos fundamentais não pode ser alterado pela sobrecarga de operadores. O programador não pode, por exemplo, alterar o significado de como `+` soma dois inteiros. A sobrecarga de operadores funciona somente com objetos de tipos definidos pelo usuário ou com uma mistura de um objeto de um tipo definido pelo usuário e um objeto de um tipo fundamental.



Observação de engenharia de software 11.2

Pelo menos um argumento de uma função operadora deve ser um objeto ou referência de um tipo definido pelo usuário. Isso impede que os programadores alterem a maneira como os operadores funcionam sobre tipos fundamentais.



Erro comum de programação 11.4

Tentar modificar a maneira como um operador funciona com objetos de tipos fundamentais é um erro de compilação.

Operadores relacionados

Sobrecarregando um operador de atribuição e um operador de adição para permitir instruções como

```
object2 = object2 + object1;
```

não implica que o operador `+=` também seja sobrecarregado para permitir instruções como

```
object2 += object1;
```

Esse comportamento só pode ser alcançado sobrecarregando explicitamente o operador `+=` dessa classe.



Erro comum de programação 11.5

Supor que sobrecarregar um operador como + sobrecarrega operadores relacionados como += ou que sobrecarregar == sobrecarrega um operador relacionado como != pode levar a erros. Os operadores só podem ser sobrecarregados explicitamente; não há sobrecarga implícita.

11.4 Funções operadoras como membros de classe versus funções globais

As funções operadoras podem ser funções-membros ou funções globais; as funções globais são freqüentemente feitas friends por razões de desempenho. As funções-membro utilizam o ponteiro `this` implicitamente para obter um de seus argumentos de objeto de classe (o operando esquerdo para operadores binários). Os argumentos para ambos os operandos de um operador binário devem ser explicitamente listados em uma chamada de função global.

Operadores que devem ser sobrecarregados como funções-membro

Ao sobrecarregar `()`, `[]`, `->` ou qualquer um dos operadores de atribuição, a função de sobrecarga de operadores devem ser declarada como um membro de classe. Para os outros operadores, as funções de sobrecarga de operadores podem ser membros de classe ou funções globais.

Operadores como funções-membro e funções globais

Se uma função operadora é implementada como uma função-membro ou como uma função global, o operador ainda é utilizado da mesma maneira em expressões. Então qual implementação é melhor?

Quando uma função operadora é implementada como uma função-membro, o operando na extrema esquerda (ou único) deve ser um objeto (ou uma referência a um objeto) da classe do operador. Se o operando esquerdo precisar ser um objeto de uma classe diferente ou um tipo fundamental, essa função operadora deve ser implementada como uma função global (como faremos na Seção 11.5 quando sobrecarregarmos `<<` e `>>` como os operadores de inserção de fluxo e extração de fluxo, respectivamente). Uma função operadora global pode se tornar um friend de uma classe se essa função precisar acessar membros `private` ou `protected` dessa classe diretamente.

As funções-membro do operador de uma classe específica são chamadas (implicitamente pelo compilador) somente quando o operando esquerdo de um operador binário for especificamente um objeto dessa classe ou quando o único operando de um operador unário for um objeto dessa classe.

Por que os operadores sobrecarregados de inserção e de extração de fluxo são sobrecarregados como funções globais

O operador de inserção de fluxo (`<<`) sobrecarregado é utilizado em uma expressão em que o operando esquerdo tem o tipo `ostream &`, como em `cout << classObject`. Para utilizar o operador dessa maneira, em que o operando direito é um objeto de uma classe definida pelo usuário, ele deve ser sobrecarregado como uma função global. Para ser uma função-membro, o operador `<<` teria de ser membro da classe `ostream`. Isso não é possível para classes definidas pelo usuário, visto que não temos permissão de modificar as classes da C++ Standard Library. De modo semelhante, o operador de extração de fluxo (`>>`) sobrecarregado é utilizado em uma expressão em que o operando esquerdo tem o tipo `istream &`, como em `cin >> classObject`, e o operando direito é um objeto de uma classe definida pelo usuário, portanto ele também deve ser uma função global. Além disso, cada uma dessas funções operadoras sobrecarregadas pode requerer acesso aos membros de dados `private` do objeto de classe cuja saída ou entrada está sendo feita, portanto podemos transformar essas funções operadoras sobrecarregadas em funções friend da classe, por razões de desempenho.



Dica de desempenho 11.1

É possível sobrecarregar um operador como uma função não-friend global, mas essa função que requer acesso aos dados `private` ou `protected` de uma classe precisaria utilizar as funções `set` ou `get` fornecidas na interface `public` dessa classe. O overhead de chamar essas funções poderia causar um desempenho ruim, desse modo essas funções podem ser colocadas inline para melhorar o desempenho.

Operadores comutativos

Outra razão pela qual poderíamos preferir uma função global a sobrecarregar um operador é permitir que o operador seja comutativo. Por exemplo, suponha que temos um objeto, `number`, do tipo `long int`, e um objeto `bigInteger1`, da classe `HugeInteger` (uma classe em que os inteiros podem ser arbitrariamente grandes em vez de limitados pelo tamanho da palavra de máquina do hardware subjacente; a classe `HugeInteger` é desenvolvida nos exercícios do capítulo). O operador de adição (+) produz um objeto `HugeInteger` temporário como a soma de um `HugeInteger` e um `long int` (como na expressão `bigInteger1 + number`) ou como a soma de um `long int` e um `HugeInteger` (como na expressão `number + bigInteger1`). Portanto, requeremos que o operador de adição seja comutativo (exatamente como ele é com dois operandos do tipo fundamental). O problema é que o objeto de classe deve aparecer à esquerda do operador de adição se este precisar ser sobrecarregado como uma função-membro. Então, sobrecarregamos o operador como uma função global para permitir que o `HugeInteger` apareça à direita da adição. A função `operator+`, que lida com o `HugeInteger` à esquerda, ainda pode ser uma função-membro.

11.5 Sobrecarregando operadores de inserção e extração de fluxo

O C++ é capaz de realizar a entrada e a saída dos tipos fundamentais utilizando o operador de extração de fluxo >> e o operador de inserção de fluxo <<. As bibliotecas de classes fornecidas com os compiladores C++ sobrecarregam esses operadores para processar todos os tipos fundamentais, inclusive os ponteiros e strings `char *` no estilo C. Os operadores de inserção de fluxo e de extração de fluxo também podem ser sobrecarregados para realizar entrada e saída de tipos definidos pelo usuário. O programa das figuras 11.3–11.5 demonstra como sobrecarregar esses operadores para tratar dados de uma classe de número de telefone definida pelo usuário chamada `PhoneNumber`. Esse programa supõe que os números de telefone são inseridos corretamente.

A função do operador de extração de fluxo `operator>>` (Figura 11.4, linhas 22–31) aceita a referência `istream input` e a referência `PhoneNumber num` como argumentos e retorna uma referência `istream`. A função operadora `operator>>` realiza a entrada de números de telefone na forma

```
(800) 555-1212
```

em objetos da classe `PhoneNumber`. Quando o compilador vê a expressão

```
cin >> phone
```

na linha 19 da Figura 11.5, ele gera a chamada de função global

```
operator>>( cin, phone );
```

Quando essa chamada executa, o parâmetro de referência `input` (Figura 11.4, linha 22) se torna um alias para `cin` e o parâmetro de referência `number` se torna um alias para `phone`. A função operadora lê como strings as três partes do número de telefone nos membros `areaCode` (linha 25), `exchange` (linha 27) e `line` (linha 29) do objeto referenciado `PhoneNumber` pelo parâmetro `number`. O manipulador de fluxo `setw` limita o número de caracteres lidos em cada array de caractere. Quando utilizado com `cin` e strings, `setw` restringe o número de caracteres lidos ao número de caracteres especificados por seu argumento (isto é, `setw( 3 )` permite que três caracteres sejam lidos). Os caracteres de parênteses, espaço e traço são ignorados chamando a função-membro `istream ignore` (Figura 11.4, linhas 24, 26 e 28), que descarta o número especificado de caracteres no fluxo de entrada (um caractere por padrão). A função `operator>>` retorna `istream` à referência `input` (isto é, `cin`). Isso permite que as operações de entrada em objetos `PhoneNumber` sejam colocadas em cascata com operações de entrada sobre outros objetos `PhoneNumber` ou sobre objetos de outros tipos de dados. Por exemplo, um programa pode inserir dois objetos `PhoneNumber` em uma instrução como esta:

```
cin >> phone1 >> phone2;
```

Primeiro, a expressão `cin >> phone1` executa fazendo a chamada de função global

```
operator>>( cin, phone1 );
```

```
1 // Figura 11.3: PhoneNumber.h
2 // definição da classe PhoneNumber
3 #ifndef PHONENUMBER_H
4 #define PHONENUMBER_H
5
6 #include <iostream>
7 using std::ostream;
8 using std::istream;
9
10 #include <string>
11 using std::string;
12
13 class PhoneNumber
14 {
15     friend ostream &operator<<( ostream &, const PhoneNumber & );
16     friend istream &operator>>( istream &, PhoneNumber & );
17 private:
18     string areaCode; // código de área (de cidade) de 3 algarismos
19     string exchange; // prefixo (de bairro/região) de 3 algarismos
20     string line; // linha de 4 algarismos
21 }; // fim da classe PhoneNumber
22
23 #endif
```

Figura 11.3 Classe `PhoneNumber` com operadores de inserção de fluxo e de extração de fluxo sobrecarregados como funções `friend`.

```

1 // Figura 11.4: PhoneNumber.cpp
2 // Operadores de inserção de fluxo e de extração de fluxo sobrecarregados
3 // para a classe PhoneNumber.
4 #include <iomanip>
5 using std::setw;
6
7 #include "PhoneNumber.h"
8
9 // operador de inserção de fluxo sobrecarregado; não pode ser
10 // uma função-membro se quiséssemos invocá-lo com
11 // cout << somePhoneNumber;
12 ostream &operator<< ( ostream &output, const PhoneNumber &number )
13 {
14     output << "(" << number.areaCode << " ) "
15         << number.exchange << "-" << number.line;
16     return output; // permite cout << a << b << c;
17 } // fim da função operator<<
18
19 // operador de extração de fluxo sobrecarregado; não pode ser
20 // uma função-membro se quiséssemos invocá-lo com
21 // cin >> somePhoneNumber;
22 istream &operator>> ( istream &input, PhoneNumber &number )
23 {
24     input.ignore(); // pula (
25     input >> setw( 3 ) >> number.areaCode; // entrada do código de área
26     input.ignore( 2 ); // pula ) e espaço
27     input >> setw( 3 ) >> number.exchange; // entrada do prefixo (exchange)
28     input.ignore(); // pula traço (-)
29     input >> setw( 4 ) >> number.line; // entrada de linha
30     return input; // permite cin >> a >> b >> c;
31 } // fim da função operator>>

```

Figura 11.4 Os operadores de inserção e de extração de fluxo sobrecarregados para a classe PhoneNumber.

```

1 // Figura 11.5: fig11_05.cpp
2 // Demonstrando os operadores de inserção
3 // e extração de fluxo sobrecarregados da classe PhoneNumber.
4 #include <iostream>
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include "PhoneNumber.h"
10
11 int main()
12 {
13     PhoneNumber phone; // cria objeto phone
14
15     cout << "Enter phone number in the form (123) 456-7890:" << endl;
16
17     // cin >> phone invoca operator>> emitindo implicitamente
18     // a chamada da função global operator>>( cin, phone )
19     cin >> phone;
20

```

Figura 11.5 Operadores de inserção e extração de fluxo sobrecarregados.

(continua)


```

21     cout << "The phone number entered was: ";
22
23     // cout << phone invoca operator<< emitindo implicitamente
24     // chamada da função global operator<<( cout, telefone )
25     cout << phone << endl;
26     return 0;
27 } // fim de main

```

Enter phone number in the form (123) 456-7890:
(800) 555-1212
 The phone number entered was: (800) 555-1212

Figura 11.5 Operadores de inserção e extração de fluxo sobrecarregados.

(continuação)

Essa chamada então retorna uma referência a `cin` como o valor de `cin >> phone1`, de modo que a parte restante da expressão é interpretada simplesmente como `cin >> phone2`. Isso executa fazendo a chamada de função global

```
operator>>( cin, phone2 );
```

A função operadora de inserção de fluxo (Figura 11.4, linhas 12–17) aceita uma referência `ostream` (output) e uma referência `const PhoneNumber` (`number`) como argumentos e retorna uma referência `ostream`. A função `operator<<` exibe objetos do tipo `PhoneNumber`. Quando o compilador vê a expressão

```
cout << phone
```

na linha 25 da Figura 11.5, ele gera a chamada de função global

```
operator<<( cout, phone );
```

A função `operator<<` exibe as partes do número de telefone como strings, porque elas são armazenadas como objetos `string`.



Dica de prevenção de erro 11.1

Retornar uma referência a partir de uma função operadora << ou >> sobrecarregada é normalmente bem-sucedido, porque `cout`, `cin` e a maioria dos objetos de fluxo são globais ou pelo menos de longa duração. Retornar uma referência a uma variável automática ou outro objeto temporário é perigoso — pois cria ‘referências oscilantes’ a objetos não-existentes.

Observe que as funções `operator>>` e `operator<<` são declaradas em `PhoneNumber` como funções `friend` globais (Figura 11.3, linhas 15–16). Elas são funções globais porque os objetos da classe `PhoneNumber` aparecem em cada caso como o operando direito do operador. Lembre-se de que as funções operadoras sobrecarregadas para operadores binários só podem ser funções-membros quando o operando esquerdo for um objeto da classe em que a função é membro. Os operadores de entrada e saída sobrecarregados são declarados como `friends` se precisam acessar membros de classe não-`public` diretamente por razões de desempenho ou porque a classe não pode oferecer as funções `get` apropriadas. Observe também que a referência `PhoneNumber` na lista de parâmetros da função `operator<<` (Figura 11.4, linha 12) é `const`, porque `PhoneNumber` será simplesmente enviado para a saída, e a referência `PhoneNumber` na lista de parâmetros da função `operator>>` (linha 22) é não-`const`, porque o objeto `PhoneNumber` deve ser modificado para armazenar a entrada do número de telefone no objeto.



Observação de engenharia de software 11.3

Novas capacidades de entrada/saída para tipos definidos pelo usuário são adicionadas ao C++ sem modificar as classes de entrada/saída da biblioteca-padrão do C++. Esse é outro exemplo da extensibilidade da linguagem de programação C++.

11.6 Sobrecarregando operadores unários

Um operador unário de uma classe pode ser sobrecarregado como uma função-membro não-`static` sem argumentos ou como uma função global com um argumento; esse argumento deve ser um objeto da classe ou uma referência a um objeto da classe. As funções-membro que implementam os operadores sobrecarregados devem ser não-`static` para que possam acessar os dados não-`static` em cada objeto da classe. Lembre-se de que as funções-membro `static` podem acessar apenas os membros de dados `static` da classe.

Mais adiante neste capítulo, sobrecarregaremos o operador unário `!` para testar se um objeto da classe `String` que criamos (Seção 11.10) está vazio e retornar um resultado `bool`. Considere a expressão `!s`, em que `s` é um objeto da classe `String`. Quando um operador unário como `!` é sobrecarregado como uma função-membro sem argumentos e o compilador vê a expressão `!s`, o compilador gera a chamada `s.operator!()`. O operando `s` é o objeto de classe para o qual a função-membro `operator!` da classe `String` está sendo invocada. A função é declarada na definição de classe como mostrado a seguir:

```
class String
{
public:
    bool operator!() const;
    ...
}; // fim da classe String
```

Um operador unário como ! pode ser sobrecarregado como uma função global com um argumento de duas maneiras diferentes — com um argumento que seja um objeto (isso requer uma cópia do objeto, para que os efeitos colaterais da função não sejam aplicados ao objeto original), ou com um argumento que seja uma referência a um objeto (nenhuma cópia do objeto original é feita, desse modo, todos os efeitos colaterais dessa função são aplicados ao objeto original). Se *s* for um objeto da classe *String* (ou uma referência a um objeto da classe *String*), então *!s* é tratado como se a chamada *operator!(s)* tivesse sido escrita, invocando a função *operator!* global que é declarada como segue:

```
bool operator!( const String & );
```

11.7 Sobrecarregando operadores binários

Um operador binário é sobrecarregado como uma função-membro não-*static* com um argumento ou como uma função global com dois argumentos (um desses argumentos deve ser um objeto de classe ou uma referência a um objeto de classe).

Mais adiante neste capítulo, sobrecarregaremos < para comparar dois objetos *String*. Ao sobrecarregar o operador binário < como uma função-membro não-*static* de uma classe *String* com um argumento, se *y* e *z* forem objetos da classe *String*, então *y < z* será tratado como se *y.operator<(z)* tivesse sido escrita, invocando a função-membro *operator<* declarada como segue

```
class String

public:
    bool operator<( const String & ) const;
    ...
}; // fim da classe String
```

Se o operador binário < deve ser sobrecarregado como uma função global, ele deve aceitar dois argumentos — um dos quais deve ser um objeto de classe ou uma referência a um objeto de classe. Se *y* e *z* forem objetos da classe *String* ou referências a objetos da classe *String*, então *y < z* é tratado como se a chamada *operator<(y, z)* tivesse sido escrita no programa, invocando a função global *operator<* declarada da seguinte maneira:

```
bool operator<( const String &, const String & );
```

11.8 Estudo de caso: classe Array

Arrays baseados em ponteiro apresentam diversos problemas. Por exemplo, um programa pode facilmente ‘ultrapassar’ qualquer uma das extremidades de um array, porque o C++ não verifica se os subscritos saem fora do intervalo de um array (embora o programador ainda possa fazer isso explicitamente). Os arrays de tamanho *n* devem numerar seus elementos 0, ..., *n* - 1; intervalos alternativos de subscrito não são permitidos. A entrada ou saída de um array não-char inteiro não pode ser realizada de uma vez; cada elemento do array deve ser lido ou gravado individualmente. Dois arrays não podem ser comparados significativamente com operadores de igualdade ou operadores relacionais (porque os nomes de array são simplesmente ponteiros para o local em que os arrays iniciam na memória e, naturalmente, dois arrays sempre estarão em posições da memória diferentes). Quando um array é passado para uma função de uso geral projetada para lidar com arrays de qualquer tamanho, o tamanho do array deve ser passado como um argumento adicional. Um array não pode ser atribuído a outro com o(s) operador(es) de atribuição (porque os nomes de array são ponteiros *const* e um ponteiro constante não pode ser utilizado à esquerda do operador de atribuição). Essas e outras capacidades certamente parecem ‘naturais’ para lidar com arrays, mas os arrays baseados em ponteiro não fornecem essas capacidades. Entretanto, o C++ realmente fornece maneiras de implementar essas capacidades de array pelo uso de classes e de sobrecarga de operadores.

Nesse exemplo, criamos uma poderosa classe de array que realiza a verificação de intervalo para assegurar que os subscritos permaneçam dentro dos limites do Array. A classe permite que um objeto de array seja atribuído a outro com o operador de atribuição. Os objetos da classe Array conhecem seu tamanho, portanto o tamanho não precisa ser passado separadamente como um argumento durante a passagem de um Array para uma função. A entrada ou saída de Arrays inteiros pode ser feita com os operadores de extração e de inserção de fluxo, respectivamente. As comparações do Array podem ser feitas com os operadores de igualdade == e !=.

Esse exemplo aprofundará seu conhecimento sobre abstração de dados. Provavelmente você vai querer sugerir outros aprimoramentos a essa classe Array. O desenvolvimento de classe é uma atividade interessante, criativa e intelectualmente desafiadora — sempre com o objetivo de ‘fabricar classes valiosas’.

O programa das figuras 11.6–11.8 demonstra a classe Array e seus operadores sobrecarregados. Primeiro percorremos *main* (Figura 11.8). Depois consideramos a definição de classe (Figura 11.6) e cada uma das funções-membro e definições de função *friend* da classe (Figura 11.7).

```

1 // Figura 11.6: Array.h
2 // Classe Array para armazenar arrays de inteiros.
3 #ifndef ARRAY_H
4 #define ARRAY_H
5
6 #include <iostream>
7 using std::ostream;
8 using std::istream;
9
10 class Array
11 {
12     friend ostream &operator<<( ostream &, const Array & );
13     friend istream &operator>>( istream &, Array & );
14 public:
15     Array( int = 10 ); // construtor-padrão
16     Array( const Array & ); // construtor de cópia
17     ~Array(); // destrutor
18     int getSize() const; // retorna tamanho
19
20     const Array &operator=( const Array & ); // operador de atribuição
21     bool operator==( const Array & ) const; // operador de igualdade
22
23     // operador de desigualdade; retorna o oposto do operador ==
24     bool operator!=( const Array &right ) const
25     {
26         return ! ( *this == right ); // invoca Array::operator==
27     } // fim da função operator!=
28
29     // operador subscripto de objetos não-const retorna lvalue modificável
30     int &operator[]( int );
31
32     // operador de subscripto de objetos const retorna rvalue
33     int operator[]( int ) const;
34 private:
35     int size; // tamanho do array baseado em ponteiro
36     int *ptr; // ponteiro para o primeiro elemento do array baseado em ponteiro
37 }; // fim da classe Array
38
39 #endif

```

Figura 11.6 Definição da classe Array com operadores sobrecarregados.

```

1 // Fig 11.7: Array.cpp
2 // Definições de função-membro para a classe Array
3 #include <iostream>
4 using std::cerr;
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 #include <iomanip>
10 using std::setw;
11
12 #include <cstdlib> // sai do protótipo de função

```

Figura 11.7 Definições de função-membro e função friend da classe Array.

(continua)

```

13 using std::exit;
14
15 #include "Array.h" // definição da classe Array
16
17 // construtor-padrão para a classe Array (tamanho padrão 10)
18 Array::Array( int arraySize )
19 {
20     size = ( arraySize > 0 ? arraySize : 10 ); // valida arraySize
21     ptr = new int[ size ]; // cria espaço para array baseado em ponteiro
22
23     for ( int i = 0; i < size; i++ )
24         ptr[ i ] = 0; // configura elemento do array baseado em ponteiro
25 } // fim do construtor-padrão de Array
26
27 // copia o construtor da classe Array;
28 // deve receber uma referência para impedir a recursão infinita
29 Array::Array( const Array &arrayToCopy )
30     : size( arrayToCopy.size )
31 {
32     ptr = new int[ size ]; // cria espaço para array baseado em ponteiro
33
34     for ( int i = 0; i < size; i++ )
35         ptr[ i ] = arrayToCopy.ptr[ i ]; // copia para o objeto
36 } // fim do construtor de cópia do Array
37
38 // destrutor para a classe Array
39 Array::~Array()
40 {
41     delete [] ptr; // libera espaço do array baseado em ponteiro
42 } // fim do destrutor
43
44 // retorna o número de elementos do Array
45 int Array::getSize() const
46 {
47     return size; // número de elementos em Array
48 } // fim da função getSize
49
50 // operador de atribuição sobrecarregado;
51 // retorno const evita: (a1 = a2) = a3
52 const Array &Array::operator=( const Array &right )
53 {
54     if ( &right != this ) // evita auto-atribuição:
55     {
56         // para Arrays de tamanhos diferentes, desaloca array do lado esquerdo
57         // original, então aloca o novo array à esquerda
58         if ( size != right.size )
59         {
60             delete [] ptr; // libera espaço
61             size = right.size; // redimensiona esse objeto
62             ptr = new int[ size ]; // cria espaço para a cópia do array
63         } // fim do if interno
64
65         for ( int i = 0; i < size; i++ )
66             ptr[ i ] = right.ptr[ i ]; // copia o array para o objeto
67     } // fim do if externo
68

```

Figura 11.7 Definições de função-membro e função friend da classe Array.

(continua)

```

69     return *this; // permite x = y = z, por exemplo
70 } // fim da função operator=
71
72 // determina se dois Arrays são iguais e
73 // retorna true, caso contrário retorna false
74 bool Array::operator==( const Array &right ) const
75 {
76     if ( size != right.size )
77         return false; // arrays com diferentes números de elementos
78
79     for ( int i = 0; i < size; i++ )
80         if ( ptr[ i ] != right.ptr[ i ] )
81             return false; // o conteúdo do Array não é igual
82
83     return true; // Arrays são iguais
84 } // fim da função operator==
85
86 // operador de subscrito sobrecarregado para Arrays não-const;
87 // retorno de referência cria um lvalue modificável
88 int &Array::operator[]( int subscript )
89 {
90     // verifica erro de subscrito fora do intervalo
91     if ( subscript < 0 || subscript >= size )
92     {
93         cerr << "\nError: Subscript " << subscript
94             << " out of range" << endl;
95         exit( 1 ); // termina o programa; subscrito fora do intervalo
96     } // fim do if
97
98     return ptr[ subscript ]; // retorno da referência
99 } // fim da função operator[]
100
101 // operador de subscrito sobrecarregado para Arrays const
102 // retorno de referência const cria um rvalue
103 int Array::operator[]( int subscript ) const
104 {
105     // verifica erro de subscrito fora do intervalo
106     if ( subscript < 0 || subscript >= size )
107     {
108         cerr << "\nError: Subscript " << subscript
109             << " out of range" << endl;
110         exit( 1 ); // termina o programa; subscrito fora do intervalo
111     } // fim do if
112
113     return ptr[ subscript ]; // retorna cópia desse elemento
114 } // fim da função operator[]
115
116 // operador de entrada sobrecarregado para a classe Array;
117 // entrada de valores para o Array inteiro
118 istream &operator>>( istream &input, Array &a )
119 {
120     for ( int i = 0; i < a.size; i++ )
121         input >> a.ptr[ i ];
122
123     return input; // permite cin >> x >> y;
124 } // fim da função

```

Figura 11.7 Definições de função-membro e função friend da classe Array.

(continua)

```

125
126 // operador de saída sobrecarregado para classe Array
127 ostream &operator<<( ostream &output, const Array &a )
128 {
129     int i;
130
131     // gera saída do array baseado em ptr private
132     for ( i = 0; i < a.size; i++ )
133     {
134         output << setw( 12 ) << a.ptr[ i ];
135
136         if ( ( i + 1 ) % 4 == 0 ) // 4 números por linha de saída
137             output << endl;
138     } // fim do for
139
140     if ( i % 4 != 0 ) // termina a última linha de saída
141         output << endl;
142
143     return output; // permite cout << x << y;
144 } // fim da função operator<<

```

Figura 11.7 Definições de função-membro e função friend da classe Array.

(continuação)

```

1 // Figura 11.8: fig11_08.cpp
2 // Programa de teste da classe Array.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 #include "Array.h"
9
10 int main()
11 {
12     Array integers1( 7 ); // Array de sete elementos
13     Array integers2; // Array de 10 elementos por padrão
14
15     // imprime o tamanho e o conteúdo de integers1
16     cout << "Size of Array integers1 is "
17         << integers1.getSize()
18         << "\nArray after initialization:\n" << integers1;
19
20     // imprime o tamanho e o conteúdo de integers2
21     cout << "\nSize of Array integers2 is "
22         << integers2.getSize()
23         << "\nArray after initialization:\n" << integers2;
24
25     // insere e imprime integers1 e integers2
26     cout << "\nEnter 17 integers:" << endl;
27     cin >> integers1 >> integers2;
28
29     cout << "\nAfter input, the Arrays contain:\n"
30         << "integers1:\n" << integers1
31         << "integers2:\n" << integers2;

```

Figura 11.8 Programa de teste da classe Array.

(continua)

```

32
33 // utiliza o operador de desigualdade (!=) sobrecarregado
34 cout << "\nEvaluating: integers1 != integers2" << endl;
35
36 if ( integers1 != integers2 )
37     cout << "integers1 and integers2 are not equal" << endl;
38
39 // cria Array integers3 utilizando integers1 como um
40 // inicializador; imprime tamanho e conteúdo
41 Array integers3( integers1 ); // invoca o construtor de cópia
42
43 cout << "\nSize of Array integers3 is "
44     << integers3.getSize()
45     << "\nArray after initialization:\n" << integers3;
46
47 // utiliza operador atribuição (=) sobrecarregado
48 cout << "\nAssigning integers2 to integers1:" << endl;
49 integers1 = integers2; // note que o Array alvo é menor
50
51 cout << "integers1:\n" << integers1
52     << "integers2:\n" << integers2;
53
54 // utiliza operador de igualdade (==) sobrecarregado
55 cout << "\nEvaluating: integers1 == integers2" << endl;
56
57 if ( integers1 == integers2 )
58     cout << "integers1 and integers2 are equal" << endl;
59
60 // utiliza operador de subscrito sobrecarregado para criar rvalue
61 cout << "\nintegers1[5] is " << integers1[ 5 ];
62
63 // utiliza operador de subscrito sobrecarregado para criar lvalue
64 cout << "\n\nAssigning 1000 to integers1[5]" << endl;
65 integers1[ 5 ] = 1000;
66 cout << "integers1:\n" << integers1;
67
68 // tentativa de utilizar subscrito fora do intervalo
69 cout << "\nAttempt to assign 1000 to integers1[15]" << endl;
70 integers1[ 15 ] = 1000; // ERRO: fora do intervalo
71 return 0;
72 } // fim de main

```

Size of Array integers1 is 7
Array after initialization:

0	0	0	0
0	0	0	

Size of Array integers2 is 10
Array after initialization:

0	0	0	0
0	0	0	0
0	0		

Enter 17 integers:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

Figura 11.8 Programa de teste da classe Array.

(continua)


```

After input, the Arrays contain:
integers1:
    1      2      3      4
    5      6      7
integers2:
    8      9      10     11
    12     13     14     15
    16     17

Evaluating: integers1 != integers2
integers1 and integers2 are not equal

Size of Array integers3 is 7
Array after initialization:
    1      2      3      4
    5      6      7

Assigning integers2 to integers1:
integers1:
    8      9      10     11
    12     13     14     15
    16     17
integers2:
    8      9      10     11
    12     13     14     15
    16     17

Evaluating: integers1 == integers2
integers1 and integers2 are equal

integers1[5] is 13

Assigning 1000 to integers1[5]
integers1:
    8      9      10     11
    12     1000   14     15
    16     17

Attempt to assign 1000 to integers1[15]

Error: Subscript 15 out of range

```

Figura 11.8 Programa de teste da classe Array.

(continuação)

Criando Arrays, gerando saída de seu tamanho e exibindo seu conteúdo

O programa começa instanciando dois objetos da classe Array — `integers1` (Figura 11.8, linha 12) com sete elementos, e `integers2` (Figura 11.8, linha 13) com o tamanho do Array-padrão — 10 elementos (especificado pelo protótipo do construtor-padrão Array na Figura 11.6, linha 15). As linhas 16–18 utilizam a função-membro `getSize` para determinar o tamanho de `integers1` e geram a saída de `integers1`, utilizando o operador de inserção de fluxo sobrecarregado Array. A saída de exemplo confirma que os elementos do Array foram configurados corretamente como zeros pelo construtor. Em seguida, as linhas 21–23 geram saída do tamanho do Array `integers2` e geram saída de `integers2`, utilizando o operador de inserção de fluxo sobrecarregado Array.

Utilizando o operador de inserção de fluxo sobrecarregado para preencher um Array

A linha 26 pede para o usuário inserir 17 inteiros. A linha 27 utiliza o operador de extração de fluxo sobrecarregado Array para ler esses valores para ambos os arrays. Os sete primeiros valores são armazenados em `integers1` e os 10 valores restantes são armazenados em `integers2`. As linhas 29–31 geram saída dos dois arrays com o operador de inserção de fluxo sobrecarregado Array para confirmar que a entrada foi realizada corretamente.

Utilizando o operador de desigualdade sobrecarregado

A linha 36 testa o operador de desigualdade sobrecarregado avaliando a condição

```
integers1 != integers2
```

A saída de programa mostra que os Arrays não são realmente iguais.

*Inicializando um novo **Array** com uma cópia do conteúdo de um **Array** existente*

A linha 41 instancia um terceiro Array chamado `integers3` e o inicializa com uma cópia do Array `integers1`. Isso invoca o **construtor de cópia** de Array para copiar os elementos de `integers1` em `integers3`. Discutimos os detalhes do construtor de cópia em breve. Observe que o construtor de cópia também pode ser invocado escrevendo a linha 41 da seguinte maneira:

```
Array integers3 = integers1;
```

O sinal de igual na instrução anterior *não* é o operador de atribuição. Quando um sinal de igual aparece na declaração de um objeto, ele invoca um construtor para esse objeto. Essa forma pode ser utilizada para passar somente um único argumento para um construtor.

As linhas 43–45 geram a saída do tamanho de `integers3` e a saída de `integers3`, utilizando o operador de inserção de fluxo sobrecarregado Array para confirmar que os elementos Array foram configurados corretamente pelo construtor de cópia.

Utilizando o operador de atribuição sobrecarregado

Em seguida, a linha 49 testa o operador de atribuição (=) sobrecarregado atribuindo `integers2` a `integers1`. As linhas 51–52 imprimem ambos os objetos Array para confirmar que a atribuição foi bem-sucedida. Observe que `integers1` armazenava originalmente 7 inteiros e foi redimensionado para armazenar uma cópia dos 10 elementos em `integers2`. Como veremos, o operador de atribuição sobrecarregado realiza essa operação de redimensionamento de uma maneira que é transparente ao código-cliente.

Utilizando o operador de igualdade sobrecarregado

Em seguida, a linha 57 utiliza o operador de igualdade (==) sobrecarregado para confirmar que os objetos `integers1` e `integers2` são de fato idênticos depois da atribuição.

Utilizando o operador de subscrito sobrecarregado

A linha 61 utiliza o operador de subscrito sobrecarregado para referenciar `integers1[ 5 ]` — um elemento no intervalo de `integers1`. Esse nome subscrito é utilizado como um *rvalue* para imprimir o valor armazenado em `integers1[ 5 ]`. A linha 65 utiliza `integers1[ 5 ]` como um *lvalue* modificável no lado esquerdo de uma instrução de atribuição para atribuir um novo valor, 1000, ao elemento 5 de `integers1`. Veremos que `operator[]` retorna uma referência para utilizar como o *lvalue* modificável depois que o operador confirma que 5 é um subscrito válido para `integers1`.

A linha 70 tenta atribuir o valor 1000 a `integers1[ 15 ]` — um elemento fora do intervalo. Neste exemplo, `operator[]` determina que o subscrito está fora do intervalo, imprime uma mensagem e termina o programa. Observe que destacamos em *itálico* a linha 70 do programa para enfatizar que é um erro acessar um elemento que está fora do intervalo. Esse é um erro de lógica de tempo de execução, não um erro de compilação.

Curiosamente, o operador de subscrito de array `[]` não está restrito somente ao uso de arrays; ele também pode ser utilizado, por exemplo, para selecionar elementos de outros tipos de classes contêineres, como listas vinculadas, strings e dicionários. Além disso, quando as funções `operator[]` são definidas, os subscritos não mais precisam ser inteiros — caracteres, strings, números de ponto flutuante ou até objetos de classes definidas pelo usuário também poderiam ser utilizados. No Capítulo 23, “Standard Template Library (STL)”, discutimos a classe STL `map` que permite subscritos do tipo não inteiro.

*Definição da classe **Array***

Agora que vimos como esse programa opera, vamos percorrer o cabeçalho de classe (Figura 11.6). À medida que nos referimos a cada função-membro no cabeçalho, discutimos a implementação dessa função na Figura 11.7. Na Figura 11.6, as linhas 35–36 representam os membros de dados `private` da classe Array. Cada objeto Array consiste em um membro `size` indicando o número de elementos no Array e um ponteiro `int` — `ptr` — que aponta para o array de inteiros baseado em ponteiro dinamicamente alocado gerenciado pelo objeto Array.

*Sobrecarregando operadores de inserção e extração de fluxo como **friends***

As linhas 12–13 da Figura 11.6 declaram os operadores de inserção e de extração de fluxo sobrecarregados como *friends* da classe Array. Quando o compilador vê uma expressão como `cout << arrayObject`, ele invoca a função `operator<<` global com a chamada

```
operator<<( cout, arrayObject )
```

Quando o compilador vê uma expressão como `cin >> arrayObject`, ele invoca a função `operator>>` global com a chamada

```
operator>>( cin, arrayObject )
```

Notamos novamente que essas funções operadoras de inserção e de extração de fluxo não podem ser membros da classe Array, porque o objeto Array é sempre mencionado à direita do operador de inserção de fluxo e do operador de extração de fluxo. Se essas funções operadoras precisassem ser membros da classe Array, as seguintes instruções mal construídas teriam de ser utilizadas para gerar a saída e a entrada de um Array:

```
arrayObject << cout;
arrayObject >> cin;
```

Essas instruções seriam confusas para a maioria dos programadores em C++, que estão familiarizados com `cout` e `cin` aparecendo como os operandos esquerdos de `<<` e `>>`, respectivamente.

A função `operator<<` (definida na Figura 11.7, linhas 127–144) imprime o número de elementos indicados por `size` a partir do array de inteiro para qual `ptr` aponta. A função `operator>>` (definida na Figura 11.7, linhas 118–124) realiza a entrada diretamente no array para o qual `ptr` aponta. Cada uma dessas funções operadoras retorna uma referência apropriada para permitir instruções de saída e entrada em cascata, respectivamente. Observe que cada uma dessas funções tem acesso aos dados `private` de um `Array` porque essas funções são declaradas como `friends` da classe `Array`. Além disso, observe que as funções `getSize` e `operator[]` da classe `Array` poderiam ser utilizadas por `operator<<` e `operator>>`, caso em que essas funções operadoras não precisariam ser `friends` da classe `Array`. Entretanto, as chamadas de função adicionais poderiam aumentar o overhead de tempo de execução.

Construtor-padrão `Array`

A linha 15 da Figura 11.6 declara o construtor-padrão para a classe e especifica um tamanho-padrão de 10 elementos. Quando o compilador vê uma declaração como a linha 13 na Figura 11.8, ele invoca o construtor-padrão da classe `Array` (lembre-se de que o construtor-padrão nesse exemplo realmente recebe um único argumento `int` que tem um valor-padrão de 10). O construtor-padrão (definido na Figura 11.7, linhas 18–25) valida e atribui o argumento ao membro de dados `size`, utiliza `new` para obter a memória para a representação interna baseada em ponteiro desse array e atribui o ponteiro retornado por `new` ao membro de dados `ptr`. Então o construtor usa uma instrução `for` para configurar todos os elementos do array como zero. É possível ter uma classe `Array` que não inicializa seus membros se, por exemplo, esses membros precisarem ser lidos em algum momento posterior; mas essa é considerada uma prática de programação ruim. Arrays, e objetos em geral, devem ser adequadamente inicializados e mantidos em um estado consistente.

Construtor de cópia `Array`

A linha 16 da Figura 11.6 declara um **construtor de cópia** (definido na Figura 11.7, linhas 29–36) que inicializa um `Array` fazendo uma cópia de um objeto `Array` existente. Essa cópia deve ser feita cuidadosamente para evitar a armadilha de deixar os objetos `Array` apontando para a mesma memória dinamicamente alocada. Esse é exatamente o problema que ocorreria com a cópia de membro a membro padrão, se o compilador tivesse permissão de definir um construtor de cópia-padrão para essa classe. Os construtores de cópia são invocados sempre que a cópia de um objeto for necessária, como ao passar um objeto por valor para uma função, ao retornar um objeto por valor a partir de uma função ou ao inicializar um objeto com uma cópia de outro objeto da mesma classe. O construtor de cópia é chamado em uma declaração quando um objeto da classe `Array` é instanciado e inicializado com outro objeto da classe `Array`, como na declaração da linha 41 da Figura 11.8.



Observação de engenharia de software 11.4

O argumento para um construtor de cópia deve ser uma referência `const` para permitir que um objeto `const` seja copiado.



Erro comum de programação 11.6

Observe que um construtor de cópia deve receber seu argumento por referência, não por valor. Caso contrário, a chamada do construtor de cópia resulta em recursão infinita (um erro de lógica fatal) porque receber um objeto por valor requer que o construtor de cópia faça uma cópia do objeto de argumento. Lembre-se de que a qualquer hora em que a cópia de um objeto for requerida, o construtor de cópia da classe será chamado. Se recebesse seu argumento por valor, o construtor de cópia chamaria a si mesmo recursivamente para fazer uma cópia de seu argumento!

O construtor de cópia para `Array` utiliza um inicializador de membro (Figura 11.7, linha 30) para copiar o `size` do inicializador `Array` dentro do membro de dados `size`, utiliza `new` (linha 32) para obter a memória para a representação interna baseada em ponteiro deste `Array` e atribui o ponteiro retornado por `new` ao membro de dados `ptr`.¹ Em seguida, o construtor de cópia utiliza uma instrução `for` para copiar todos os elementos do inicializador `Array` dentro do novo objeto `Array`. Observe que um objeto de uma classe pode ver os dados `private` de qualquer outro objeto dessa classe (utilizando um `handle` que indica que objeto acessar).



Erro comum de programação 11.7

*Se o construtor de cópia simplesmente copiasse o ponteiro no objeto de origem para o ponteiro do objeto-alvo, então ambos os objetos apontariam para a mesma memória dinamicamente alocada. O primeiro destrutor a executar então excluiria a memória dinamicamente alocada, e o `ptr` do outro objeto seria indefinido, uma situação chamada de **ponteiro oscilante** — que provavelmente resultaria em um grave erro de tempo de execução (como a terminação precoce do programa) quando o ponteiro fosse utilizado.*

¹ Observe que `new` poderia não conseguir obter a memória necessária. Lidamos com as falhas `new` no Capítulo 16, “Tratamento de exceções”.

Destrutor Array

A linha 17 da Figura 11.6 declara o destrutor para a classe (definido na Figura 11.7, linhas 39–42). O destrutor é invocado quando um objeto da classe Array sai de escopo. O destrutor utiliza `delete []` para liberar a memória alocada dinamicamente por `new` no construtor.

Função-membro `getSize`

A linha 18 da Figura 11.6 declara a função `getSize` (definida na Figura 11.7, linhas 45–48) que retorna o número de elementos no Array.

Operador de atribuição sobrecarregado

A linha 20 da Figura 11.6 declara a função operadora de atribuição sobrecarregada para a classe. Ao ver a expressão `integers1 = integers2` na linha 49 da Figura 11.8, o compilador invoca a função-membro `operator=` com a chamada

```
integers1.operator=( integers2 )
```

A implementação da função-membro `operator=` (Figura 11.7, linhas 52–70) testa a **auto-atribuição** (linha 54) em que um objeto da classe Array está sendo atribuído a si mesmo. Quando `this` for igual ao endereço do operando `right`, uma auto-atribuição está sendo tentada, então a atribuição é ignorada (isto é, o objeto já é ele mesmo; logo veremos por que a auto-atribuição é perigosa). Se não for uma auto-atribuição, então a função-membro determina se os tamanhos dos dois arrays são idênticos (linha 58); nesse caso, o array de inteiros original no objeto Array do lado esquerdo não é realocado. Caso contrário, `operator=` utiliza `delete` (linha 60) para liberar a memória alocada originalmente para o array-alvo, copia o `size` do array de origem para o `size` do array-alvo (linha 61), utiliza `new` para alocar memória para o array-alvo e coloca o ponteiro retornado por `new` no member `ptr` do array.² Então a instrução `for` nas linhas 65–66 copia os elementos do array de origem para o array-alvo. Independentemente de essa ser ou não uma auto-atribuição, a função-membro retorna o objeto atual (isto é, `*this` na linha 69) como uma referência constante; isso permite atribuições de Array em cascata como `x = y = z`. Se ocorresse a auto-atribuição e as funções `operator=` não testassem esse caso, `operator=` excluiria a memória dinâmica associada com o objeto Array antes que a atribuição estivesse completa. Isso deixaria `ptr` apontando para a memória que foi desalocada, o que poderia levar a erros fatais de tempo de execução.



Observação de engenharia de software 11.5

Um construtor de cópia, um destrutor e um operador de atribuição sobrecarregado normalmente são fornecidos como um grupo para qualquer classe que utiliza memória dinamicamente alocada.



Erro comum de programação 11.8

Não fornecer um operador de atribuição sobrecarregado e um construtor de cópia para uma classe quando os objetos dessa classe contêm ponteiros para memória dinamicamente alocada é um erro de lógica.



Observação de engenharia de software 11.6

É possível impedir que um objeto de uma classe seja atribuído a outra. Isso é feito declarando o operador de atribuição como um membro `private` da classe.



Observação de engenharia de software 11.7

É possível impedir que objetos de classe sejam copiados; para fazer isso, simplesmente torne ambos, o operador de atribuição sobrecarregado e o construtor de cópia dessa classe, `private`.

Operadores de igualdade e desigualdade sobrecarregados

A linha 21 da Figura 11.6 declara o operador de igualdade (`==`) sobrecarregado para a classe. Ao ver a expressão `integers1 == integers2` na linha 57 da Figura 11.8, o compilador invoca a função-membro `operator==` com a chamada

```
integers1.operator==( integers2 )
```

A função-membro `operator==` (definida na Figura 11.7, linhas 74–84) retorna `false` imediatamente se os membros `size` dos arrays não forem iguais. Caso contrário, `operator==` compara cada par de elementos. Se todos eles forem iguais, a função retorna `true`. O primeiro par de elementos que diferir faz com que a função retorne `false` imediatamente.

As linhas 24–27 do arquivo de cabeçalho definem o operador de desigualdade (`!=`) sobrecarregado para a classe. A função-membro `operator!=` usa a função `operator==` sobrecarregada para determinar se um Array é igual a outro, então retorna o oposto desse resultado.

² Mais uma vez, `new` poderia falhar. Discutimos as falhas de `new` no Capítulo 16.

Escrever `operator!=` dessa maneira permite ao programador reutilizar `operator==`, o que reduz a quantidade de código que deve ser escrita na classe. Além disso, observe que a definição de função completa para `operator!=` está no arquivo de cabeçalho `Array`. Isso permite ao compilador colocar inline a definição de `operator!=` para eliminar o overhead da chamada de função extra.

Operadores de subscrito sobrecarregados

As linhas 30 e 33 da Figura 11.6 declaram dois operadores de subscrito sobrecarregados (definidos na Figura 11.7 nas linhas 88–99 e 103–114, respectivamente). Ao ver a expressão `integers1[ 5 ]` (Figura 11.8, linha 61), o compilador invoca a função-membro sobrecarregada `operator[]` apropriada gerando a chamada

```
integers1.operator[]( 5 )
```

O compilador cria uma chamada para a versão `const` de `operator[]` (Figura 11.7, linhas 103–114) quando o operador de subscrito é utilizado em um objeto `const` `Array`. Por exemplo, se o objeto `const z` for instanciado com a instrução

```
const Array z( 5 );
```

então a versão `const` de `operator[]` é requerida para executar uma instrução como

```
cout << z[ 3 ] << endl;
```

Lembre-se, um programa pode invocar somente as funções-membro `const` de um objeto `const`.

Cada definição de `operator[]` determina se o subscrito que ele recebe como um argumento está no intervalo. Se não estiver, cada função imprime uma mensagem de erro e termina o programa com uma chamada para a função `exit` (cabeçalho `<cstdlib>`).³ Se o subscrito estiver no intervalo, a versão não-`const` de `operator[]` retorna o elemento do array apropriado como uma referência para que ele possa ser utilizado como um *lvalue* modificável (por exemplo, no lado esquerdo de uma instrução de atribuição). Se o subscrito estiver no intervalo, a versão `const` de `operator[]` retorna uma cópia do elemento apropriado do array. O caractere retornado é um *rvalue*.

11.9 Convertendo entre tipos

A maioria dos programas processa informações de muitos tipos. Às vezes, todas as operações ‘permanecem dentro de um tipo’. Por exemplo, adicionar um `int` a um `int` produz um `int` (contanto que o resultado não seja muito grande para ser representado como um `int`). Entretanto, freqüentemente é necessário converter dados de um tipo em dados de outro tipo. Isso pode acontecer em atribuições, em cálculos, na passagem de valores para funções e no retorno de valores a partir de funções. O compilador sabe executar certas conversões entre tipos fundamentais (como discutimos no Capítulo 6). Os programadores podem usar operadores de coerção para forçar conversões entre tipos fundamentais.

Mas o que dizer dos tipos definidos pelo usuário? O compilador não pode saber antecipadamente como converter entre tipos definidos pelo usuário, e entre tipos definidos pelo usuário e tipos fundamentais, portanto o programador deve especificar como fazer isso. Tais conversões podem ser realizadas com **construtores de conversão** — construtores de um único argumento que transformam objetos de outros tipos (incluindo tipos fundamentais) em objetos de uma classe particular. Na Seção 11.10, utilizamos um construtor de conversão para converter strings `char *` comuns em objetos da classe `String`.

Um **operador de conversão** (também chamado **operador de coerção**) pode ser utilizado para converter um objeto de uma classe em um objeto de outra classe ou em um objeto de um tipo fundamental. Esse operador de conversão deve ser uma função-membro não-`static`. O protótipo de função

```
A::operator char *() const;
```

declara uma função operadora de coerção sobrecarregada para converter um objeto de tipo definido pelo usuário `A` em um objeto `char *` temporário. A função operadora é declarada `const` porque não modifica o objeto original. Uma **função operadora de coerção** sobrecarregada não especifica um tipo de retorno — o tipo de retorno é o tipo no qual o objeto está sendo convertido. Se `s` é um objeto de classe, ao ver a expressão `static_cast< char * >( s )`, o compilador gera a chamada

```
s.operator char *()
```

O operando `s` é o objeto de classe `s` para o qual a função-membro `operator char *` está sendo invocada.

As funções operadoras de coerção sobrecarregadas podem ser definidas para converter objetos de tipos definidos pelo usuário em tipos fundamentais ou em objetos de outros tipos definidos pelo usuário. Os protótipos

```
A::operator int() const;
```

```
A::operator OtherClass() const;
```

declaram as funções operadoras de coerção sobrecarregadas que podem converter um objeto do tipo definido pelo usuário `A` em um inteiro ou em um objeto do tipo definido pelo usuário `OtherClass`, respectivamente.

Uma das características interessantes dos operadores de coerção e dos construtores de conversão é que, quando necessário, o compilador pode chamar essas funções implicitamente para criar objetos temporários. Por exemplo, se um objeto `s` de uma classe `String` definida pelo usuário aparecer em um programa em um local em que um `char *` comum é esperado, como

³ Observe que é mais apropriado quando um subscrito está fora de intervalo ‘lançar uma exceção’ indicando o subscrito fora do intervalo. Então o programa pode ‘capturar’ essa exceção, processá-la e, possivelmente, continuar a execução. Consulte o Capítulo 16 para obter informações adicionais sobre exceções.


```
cout << s;
```

o compilador pode chamar a função operadora de coerção sobrecarregada `operator char *` para converter o objeto em um `char *` e utilizar o `char *` resultante na expressão. Com esse operador de coerção fornecido para nossa classe `String`, o operador de inserção de fluxo não precisa ser sobrecarregado para gerar saída de uma `String` utilizando `cout`.

11.10 Estudo de caso: classe String

Como um exercício fundamental para nosso estudo sobre sobrecarga, construiremos nossa própria classe `String` para tratar a criação e a manipulação de strings (figuras 11.9–11.11). A biblioteca C++ padrão também fornece uma classe `string` semelhante e mais robusta. Apresentamos um exemplo da classe `string` padrão na Seção 11.13 e a estudamos em detalhes no Capítulo 18. Por enquanto, utilizaremos extensamente a sobrecarga de operadores para criar nossa própria classe `String`.

Primeiro, apresentamos o arquivo de cabeçalho para a classe `String`. Discutimos os dados privados utilizados para representar o objeto `String`. Então percorremos a interface pública da classe, discutindo cada um dos serviços que a classe fornece. Discutimos as definições de função-membro para a classe `String`. Para cada uma das funções operadoras sobrecarregadas, mostramos o código no programa que invoca a função operadora sobrecarregada e fornecemos uma explicação sobre como elas funcionam.

Definição da classe *String*

Agora vamos percorrer o arquivo de cabeçalho da classe `String` na Figura 11.9. Começamos com a representação interna baseada em ponteiro de uma `String`. As linhas 55–56 declaram os membros de dados `private` da classe. Nossa classe `String` tem um campo `length`, que representa o número de caracteres na string, não incluindo o caractere nulo no final, e tem um ponteiro `sPtr` que aponta para a memória dinamicamente alocada que representa a string de caracteres.

Sobrecarregando operadores de inserção e extração de fluxo como *friends*

As linhas 12–13 (Figura 11.9) declaram a função operadora de inserção de fluxo sobrecarregada `operator<<` (definida na Figura 11.10, linhas 170–174) e a função operadora de extração de fluxo sobrecarregada `operator>>` (definida na Figura 11.10, linhas 177–183) como `friends` da classe. A implementação de `operator<<` é simples e direta. Observe que `operator>>` restringe o número total de caracteres que podem ser lidos no array `temp` a 99 com `setw` (linha 180); a posição 100 é reservada para o caractere nulo de terminação da string. [Nota: Não tivemos essa restrição para `operator>>` na classe `Array` (figuras 11.6–11.7), porque o `operator>>` dessa classe leu um elemento por vez e parou de ler os valores quando o fim do array foi alcançado. O objeto `cin` não sabe fazer isso por padrão para a entrada de arrays de caracteres.] Além disso, note o uso de `operator=` (linha 181) para atribuir a string `temp` no estilo C ao objeto `String` que `s` referencia. Essa instrução invoca o construtor de conversão para criar um objeto `String` temporário que contém a string no estilo do C; o objeto `String` temporário é, então, atribuído a `s`. Poderíamos eliminar o overhead de criar o objeto `String` temporário aqui fornecendo outro operador de atribuição sobrecarregado que recebe um parâmetro do tipo `const char *`.

Construtor de conversão *String*

A linha 15 (Figura 11.9) declara um construtor de conversão. Esse construtor (definido na Figura 11.10, linhas 22–27) aceita um argumento `const char *` (que assume o padrão da string vazia; Figura 11.9, linha 15) e inicializa um objeto `String` contendo essa mesma string de caractere. É possível imaginar qualquer **construtor de um único argumento** como um construtor de conversão. Como veremos, esses construtores são úteis quando estamos fazendo qualquer operação `String` que utiliza argumentos `char *`. O construtor de conversão pode converter uma string `char *` em um objeto `String`, que pode ser então atribuído ao objeto `String` alvo. A disponibilidade desse construtor de conversão significa que não é necessário fornecer um operador de atribuição sobrecarregado para atribuir especificamente strings de caracteres aos objetos `String`. O compilador invoca o construtor de conversão para criar um objeto `String` temporário contendo a string de caracteres; então o operador de atribuição sobrecarregado é invocado para atribuir o objeto `String` temporário a outro objeto `String`.



Observação de engenharia de software 11.8

Quando um construtor de conversão é utilizado para realizar uma conversão implícita, o C++ pode aplicar apenas uma chamada de construtor implícita (isto é, uma única conversão definida pelo usuário) para tentar atender às necessidades de outro operador sobrecarregado. O compilador não atenderá às necessidades de um operador sobrecarregado para realizar uma série de conversões implícitas definidas pelo usuário.

```
1 // Figura 11.9: String.h
2 // Definição da classe String.
3 #ifndef STRING_H
4 #define STRING_H
```

Figura 11.9 Definição da classe `String` com sobrecarga de operadores.

(continua)

```

5
6 #include <iostream>
7 using std::ostream;
8 using std::istream;
9
10 class String
11 {
12     friend ostream &operator<<( ostream &, const String & );
13     friend istream &operator>>( istream &, String & );
14 public:
15     String( const char * = "" ); // construtor de conversão/padrão
16     String( const String & ); // construtor de cópia
17     ~String(); // destrutor
18
19     const String &operator=( const String & ); // operador de atribuição
20     const String &operator+=( const String & ); // operador de concatenação
21
22     bool operator!() const; // a String está vazia?
23     bool operator==( const String & ) const; // testa s1 == s2
24     bool operator<( const String & ) const; // testa s1 < s2
25
26     // testa s1 != s2
27     bool operator!=( const String &right ) const
28     {
29         return !( *this == right );
30     } // fim da função operator!=
31
32     // testa s1 > s2
33     bool operator>( const String &right ) const
34     {
35         return right < *this;
36     } // fim da função operator>
37
38     // testa s1 <= s2
39     bool operator<=( const String &right ) const
40     {
41         return !( right < *this );
42     } // fim da função operator<=
43
44     // testa s1 >= s2
45     bool operator>=( const String &right ) const
46     {
47         return !( *this < right );
48     } // fim da função operator>=
49
50     char &operator[]( int ); // operador de subscrito (lvalue modificável)
51     char operator[]( int ) const; // operador de subscrito (rvalue)
52     String operator()( int, int = 0 ) const; // retorna uma substring
53     int getLength() const; // retorna o comprimento da string
54 private:
55     int length; // comprimento de string (sem contar terminador nulo)
56     char *sPtr; // ponteiro para iniciar string baseada em ponteiro
57
58     void setString( const char * ); // função utilitária
59 }; // fim da classe String
60
61 #endif

```

Figura 11.9 Definição da classe String com sobrecarga de operadores.

(continuação)


```

1 // Figura 11.10: String.cpp
2 // Definições de função-membro para a classe String.
3 #include <iostream>
4 using std::cerr;
5 using std::cout;
6 using std::endl;
7
8 #include <iomanip>
9 using std::setw;
10
11 #include <cstring> // protótipos de strcpy e strcat
12 using std::strcmp;
13 using std::strcpy;
14 using std::strcat;
15
16 #include <cstdlib> // sai do protótipo
17 using std::exit;
18
19 #include "String.h" // definição da classe String
20
21 // construtor de conversão (e padrão) converte char * em String
22 String::String( const char *s )
23     : length( ( s != 0 ) ? strlen( s ) : 0 )
24 {
25     cout << "Conversion (and default) constructor: " << s << endl;
26     setString( s ); // chama função utilitária
27 } // fim do construtor de conversão String
28
29 // construtor de cópia
30 String::String( const String &copy )
31     : length( copy.length )
32 {
33     cout << "Copy constructor: " << copy.sPtr << endl;
34     setString( copy.sPtr ); // chama função utilitária
35 } // fim do construtor de cópia String
36
37 // Destrutor
38 String::~String()
39 {
40     cout << "Destructor: " << sPtr << endl;
41     delete [] sPtr; // libera memória de string baseada em ponteiro
42 } // fim do destrutor ~String
43
44 // operador = sobrecarregado; evita auto-atribuição
45 const String &String::operator=( const String &right )
46 {
47     cout << "operator= called" << endl;
48
49     if ( &right != this ) // evita auto-atribuição
50     {
51         delete [] sPtr; // impede vazamento de memória
52         length = right.length; // novo comprimento de String
53         setString( right.sPtr ); // chama função utilitária
54     } // fim do if
55     else
56         cout << "Attempted assignment of a String to itself" << endl;
57

```

Figura 11.10 Definições de função-membro e função friend da classe String.

(continua)

```

58     return *this; // permite atribuições em cascata
59 } // fim da função operator=
60
61 // concatena operando direito com esse objeto e armazena nesse objeto
62 const String &String::operator+=( const String &right )
63 {
64     size_t newLength = length + right.length; // novo comprimento
65     char *tempPtr = new char[ newLength + 1 ]; // cria memória
66
67     strcpy( tempPtr, sPtr ); // copia sPtr
68     strcpy( tempPtr + length, right.sPtr ); // copia right.sPtr
69
70     delete [] sPtr; // reivindica espaço antigo
71     sPtr = tempPtr; // atribui novo array a sPtr
72     length = newLength; // atribui novo comprimento a length
73     return *this; // permite chamadas em cascata
74 } // fim da função operator+=
75
76 // esta String está vazia?
77 bool String::operator!() const
78 {
79     return length == 0;
80 } // fim da função operator!
81
82 // esta String é igual à String direita?
83 bool String::operator==( const String &right ) const
84 {
85     return strcmp( sPtr, right.sPtr ) == 0;
86 } // fim da função operator==
87
88 // esta String é menor que a String direita?
89 bool String::operator<( const String &right ) const
90 {
91     return strcmp( sPtr, right.sPtr ) < 0;
92 } // fim da função operator<
93
94 // retorna a referência ao caractere na String como um lvalue modificável
95 char &String::operator[]( int subscript )
96 {
97     // testa se o subscrito está fora do intervalo
98     if ( subscript < 0 || subscript >= length )
99     {
100         cerr << "Error: Subscript " << subscript
101             << " out of range" << endl;
102         exit( 1 ); // termina o programa
103     } // fim do if
104
105     return sPtr[ subscript ]; // retorno não-const; lvalue modificável
106 } // fim da função operator[]
107
108 // retorna referência a caractere em String como rvalue
109 char String::operator[]( int subscript ) const
110 {
111     // testa se o subscrito está fora de intervalo
112     if ( subscript < 0 || subscript >= length )
113     {

```

Figura 11.10 Definições de função-membro e função friend da classe String.

(continua)

```

114     cerr << "Error: Subscript " << subscript
115         << " out of range" << endl;
116     exit( 1 ); // termina o programa
117 } // fim do if
118
119     return sPtr[ subscript ]; // retorna cópia desse elemento
120 } // fim da função operator[]
121
122 // retorna uma substring que começa em index e tem comprimento de subLength
123 String String::operator()( int index, int subLength ) const
124 {
125     // se o índice estiver fora do intervalo ou o comprimento de substring < 0,
126     // retorna um objeto String vazio
127     if ( index < 0 || index >= length || subLength < 0 )
128         return ""; // convertido em um objeto String automaticamente
129
130     // determina o comprimento da substring
131     int len;
132
133     if ( ( subLength == 0 ) || ( index + subLength > length ) )
134         len = length - index;
135     else
136         len = subLength;
137
138     // aloca array temporário para a substring e
139     // caractere de terminação nulo
140     char *tempPtr = new char[ len + 1 ];
141
142     // copia a substring para o array char e termina a string
143     strncpy( tempPtr, &sPtr[ index ], len );
144     tempPtr[ len ] = '\0';
145
146     // cria objeto String temporário contendo a substring
147     String tempString( tempPtr );
148     delete [] tempPtr; // exclui o array temporário
149     return tempString; // retorna cópia da String temporária
150 } // fim da função operator()
151
152 // retorna o comprimento da string
153 int String::getLength() const
154 {
155     return length;
156 } // fim da função getLength
157
158 // função utilitária chamada por construtores e operator=
159 void String::setString( const char *string2 )
160 {
161     sPtr = new char[ length + 1 ]; // aloca memória
162
163     if ( string2 != 0 ) // se string2 não for um ponteiro nulo, copia o conteúdo
164         strcpy( sPtr, string2 ); // cópia literal para objeto
165     else // se string2 é um ponteiro nulo, torna essa string uma string vazia
166         sPtr[ 0 ] = '\0'; // string vazia
167 } // fim da função setString
168
169 // operador de saída sobrecarregado

```

Figura 11.10 Definições de função-membro e função friend da classe String.

(continua)

```

170 ostream &operator<<( ostream &output, const String &s )
171 {
172     output << s.sPtr;
173     return output; // permite cascadeamento
174 } // fim da função operator<<
175
176 // operador de entrada sobrecarregado
177 istream &operator>>( istream &input, String &s )
178 {
179     char temp[ 100 ]; // buffer para armazenar entrada
180     input >> setw( 100 ) >> temp;
181     s = temp; // utiliza operador de atribuição da classe String
182     return input; // permite cascadeamento
183 } // fim da função operator>>

```

Figura 11.10 Definições de função-membro e função friend da classe String.

(continuação)

```

1 // Figura 11.11: fig11_11.cpp
2 // Programa de teste da classe String.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::boolalpha;
7
8 #include "String.h"
9
10 int main()
11 {
12     String s1( "happy" );
13     String s2( " birthday" );
14     String s3;
15
16     // testa operadores igualdade e relacionais sobrecarregados
17     cout << "s1 is \"" << s1 << "\"; s2 is \"" << s2
18         << "\"; s3 is \"" << s3 << "\"\n";
19     << boolalpha << "\n\nThe results of comparing s2 and s1:"
20     << "\ns2 == s1 yields " << ( s2 == s1 )
21     << "\ns2 != s1 yields " << ( s2 != s1 )
22     << "\ns2 > s1 yields " << ( s2 > s1 )
23     << "\ns2 < s1 yields " << ( s2 < s1 )
24     << "\ns2 >= s1 yields " << ( s2 >= s1 )
25     << "\ns2 <= s1 yields " << ( s2 <= s1 );
26
27
28     // testa operador de String sobrecarregado vazio (!)
29     cout << "\n\nTesting !s3:" << endl;
30
31     if ( !s3 )
32     {
33         cout << "s3 is empty; assigning s1 to s3;" << endl;
34         s3 = s1; // testa a atribuição sobrecarregada
35         cout << "s3 is \"" << s3 << "\"\n";
36     } // fim do if
37

```

Figura 11.11 Programa de teste da classe String.

(continua)

```

38 // testa o operador de concatenação de String sobrecarregado
39 cout << "\n\ns1 += s2 yields s1 = ";
40 s1 += s2; // testa concatenação sobrecarregada
41 cout << s1;
42
43 // testa o construtor de conversão
44 cout << "\n\ns1 += \" to you\" yields" << endl;
45 s1 += " to you"; // testa o construtor de conversão
46 cout << "s1 = " << s1 << "\n\n";
47
48 // testa o operador de chamada de função sobrecarregado () para a substring
49 cout << "The substring of s1 starting at\n"
50 << "location 0 for 14 characters, s1(0, 14), is:\n"
51 << s1( 0, 14 ) << "\n\n";
52
53 // testa a opção de substring "to-end-of-String"
54 cout << "The substring of s1 starting at\n"
55 << "location 15, s1(15), is: "
56 << s1( 15 ) << "\n\n";
57
58 // testa o construtor de cópia
59 String *s4Ptr = new String( s1 );
60 cout << "\n*s4Ptr = " << *s4Ptr << "\n\n";
61
62 // testa o operador de atribuição (=) com a auto-atribuição
63 cout << "assigning *s4Ptr to *s4Ptr" << endl;
64 *s4Ptr = *s4Ptr; // testa a atribuição sobrecarregada
65 cout << "*s4Ptr = " << *s4Ptr << endl;
66
67 // testa o destrutor
68 delete s4Ptr;
69
70 // testa o uso do operador de subscripto para criar um lvalue modificável
71 s1[ 0 ] = 'H';
72 s1[ 6 ] = 'B';
73 cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "
74 << s1 << "\n\n";
75
76 // testa o subscripto fora do intervalo
77 cout << "Attempt to assign 'd' to s1[30] yields:" << endl;
78 s1[ 30 ] = 'd'; // ERRO: subscripto fora do intervalo
79 return 0;
80 } // fim de main

```

Conversion (and default) constructor: happy
 Conversion (and default) constructor: birthday
 Conversion (and default) constructor:
 s1 is "happy"; s2 is " birthday"; s3 is ""

The results of comparing s2 and s1:
 s2 == s1 yields false
 s2 != s1 yields true
 s2 > s1 yields false
 s2 < s1 yields true
 s2 >= s1 yields false
 s2 <= s1 yields true

Figura 11.11 Programa de teste da classe String.

(continua)

```

Testing !s3:
s3 is empty; assigning s1 to s3;
operator= called
s3 is "happy"

s1 += s2 yields s1 = happy birthday

s1 += " to you" yields
Conversion (and default) constructor:  to you
Destructor:  to you
s1 = happy birthday to you

Conversion (and default) constructor: happy birthday
Copy constructor: happy birthday
Destructor: happy birthday
The substring of s1 starting at
location 0 for 14 characters, s1(0, 14), is:
happy birthday

Destructor: happy birthday
Conversion (and default) constructor: to you
Copy constructor: to you
Destructor: to you
The substring of s1 starting at
location 15, s1(15), is: to you

Destructor: to you
Copy constructor: happy birthday to you

*s4Ptr = happy birthday to you

assigning *s4Ptr to *s4Ptr
operator= called
Attempted assignment of a String to itself
*s4Ptr = happy birthday to you
Destructor: happy birthday to you

s1 after s1[0] = 'H' and s1[6] = 'B' is: Happy Birthday to you

Attempt to assign 'd' to s1[30] yields:
Error: Subscript 30 out of range

```

Figura 11.11 Programa de teste da classe String.

(continuação)

O construtor de conversão String poderia ser invocado nessa declaração como `String s1( "happy" )`. O construtor de conversão calcula o comprimento de seu argumento de string de caracteres e o atribui ao membro de dados `length` na lista de inicializadores de membro. Então, a linha 26 chama a função utilitária `setString` (definida na Figura 11.10, linhas 159–167), que utiliza `new` para alocar uma quantidade suficiente de memória para os membros de dados `private sPtr` e utiliza `strcpy` a fim de copiar a string de caracteres para a memória para a qual `sPtr` aponta.⁴

⁴ Há uma questão sutil na implementação desse construtor de conversão. Quando implementado, se um ponteiro nulo (isto é, 0) for passado para o construtor, o programa falhará. A maneira adequada de implementar esse construtor seria detectar se o argumento de construtor é um ponteiro nulo e, então, ‘lançar uma exceção’. O Capítulo 16 discute como podemos tornar as classes mais robustas dessa maneira. Além disso, observe que um ponteiro nulo (0) não é o mesmo que uma string vazia (""). Um ponteiro nulo é um ponteiro que não aponta para nada. Uma string vazia é uma string real que contém um único caractere nulo ('\\0').

Construtor de cópia **String**

A linha 16 na Figura 11.9 declara um construtor de cópia (definido na Figura 11.10, linhas 30–35) que inicializa um objeto `String` fazendo a cópia de um objeto `String` existente. Como com a nossa classe `Array` (figuras 11.6–11.7), essa cópia deve ser feita cuidadosamente para evitar a armadilha em que ambos os objetos `String` apontam para a mesma memória dinamicamente alocada. O construtor de cópia opera de maneira semelhante ao construtor de conversão, exceto pelo fato de que simplesmente copia o membro `length` do objeto `String` de origem para o objeto `String` alvo. Observe que o construtor de cópia chama `setString` para criar o novo espaço para a string de caracteres interna do objeto-alvo. Se simplesmente copiasse o `sPtr` do objeto origem para o `sPtr` do objeto-alvo, então ambos os objetos apontariam para mesma memória dinamicamente alocada. O primeiro destrutor a executar então excluiria a memória dinamicamente alocada e o `sPtr` de outro objeto seria indefinido (isto é, `sPtr` seria um ponteiro oscilante), uma situação potencial para causar um erro sério de tempo de execução.

Destrutor **String**

A linha 17 da Figura 11.9 declara o destrutor `String` (definido na Figura 11.10, linhas 38–42). O destrutor utiliza `delete []` para liberar a memória dinâmica para o qual `sPtr` aponta.

Operador de atribuição sobrecarregado

A linha 19 (Figura 11.9) declara a função operadora de atribuição sobrecarregada `operator=` (definida na Figura 11.10, linhas 45–59). Ao ver uma expressão como `string1 = string2`, o compilador gera a chamada de função

```
string1.operator=( string2 );
```

A função operadora de atribuição sobrecarregada `operator=` testa a auto-atribuição. Se essa for uma auto-atribuição, a função não precisará mudar o objeto. Se esse teste fosse omitido, a função excluiria imediatamente o espaço no objeto-alvo e, assim, perderia a string de caracteres, de modo que o ponteiro não apontaria para dados válidos — um exemplo clássico de um ponteiro oscilante. Se não houver auto-atribuição, a função exclui a memória e copia o campo `length` do objeto de origem para o objeto-alvo. Em seguida, `operator=` chama `setString` para criar o novo espaço para o objeto-alvo e copia a string de caracteres do objeto de origem para o objeto-alvo. Seja essa uma auto-atribuição ou não, `operator=` retorna `*this` para permitir atribuições em cascata.

Operador de atribuição de adição sobrecarregado

A linha 20 da Figura 11.9 declara o operador de concatenação de string sobrecarregado `+=` (definido na Figura 11.10, linhas 62–74). Ao ver a expressão `s1 += s2` (linha 40 da Figura 11.11), o compilador gera a chamada de função-membro

```
s1.operator+=( s2 )
```

A função `operator+=` calcula o comprimento combinado da string concatenada e a armazena na variável local `newLength`, então cria um ponteiro temporário (`tempPtr`) e aloca um novo array de caracteres no qual a string concatenada será armazenada. Em seguida, `operator+=` utiliza `strcpy` para copiar as strings de caracteres originais de `sPtr` e `right.sPtr` na memória para a qual `tempPtr` aponta. Observe que a localização em que `strcpy` copiará o primeiro caractere de `right.sPtr` é determinada pelo cálculo aritmético de ponteiro `tempPtr + length`. Esse cálculo indica que o primeiro caractere de `right.sPtr` deve ser colocado na localização `length` do array para o qual `tempPtr` aponta. Em seguida, `operator+=` utiliza `delete []` para liberar o espaço ocupado pela string de caracteres original desse objeto, atribui `tempPtr` a `sPtr` de modo que esse objeto `String` aponte para a nova string de caracteres, atribui `newLength` a `length` de modo que esse objeto `String` contenha o novo comprimento de string e retorna `*this` como uma `const String &` para permitir a cascata de operadores `+=`.

Precisamos de um segundo operador de concatenação sobrecarregado para permitir a concatenação de uma `String` e um `char *`. Não. O construtor de conversão `const char *` converte uma string no estilo C em um objeto `String` temporário, que então corresponde ao operador de concatenação sobrecarregado existente. Isso é exatamente o que o compilador faz quando encontra a linha 44 na Figura 11.11. Novamente, o C++ pode realizar essas conversões somente em um nível profundo para facilitar uma correspondência. O C++ também pode realizar uma conversão definida por compilador implícita entre tipos fundamentais antes de realizar a conversão entre um tipo fundamental e uma classe. Observe que, quando um objeto `String` temporário é criado nesse caso, o construtor de conversão e o destrutor são chamados (veja a saída resultante da linha 45, `s1 += " to you"`, na Figura 11.11). Esse é um exemplo de overhead de chamada de função que fica oculto do cliente da classe quando objetos de classe temporários são criados e destruídos durante as conversões implícitas. Overhead semelhante é gerado por construtores de cópia em passagem de parâmetros de chamada por valor e no retorno de objetos de classe por valor.



Dica de desempenho 11.2

Sobrecarregar o operador de concatenação `+=` com uma versão adicional que aceita um único argumento do tipo `const char *` torna a execução mais eficiente do que ter apenas uma versão que aceita um argumento `String`. Sem a versão `const char *` do operador `+=`, um argumento `const char *` seria primeiro convertido em um objeto `String` com o construtor de conversão da classe `String`, então o operador `+=` que recebe um argumento `String` seria chamado para realizar a concatenação.

**Observação de engenharia de software 11.9**

Utilizar conversões implícitas com operadores sobrecarregados, em vez de sobrecarregar operadores para muitos tipos diferentes de operandos, freqüentemente exige menos código, o que torna uma classe mais fácil de modificar, manter e depurar.

Operador de negação sobrecarregado

A linha 22 da Figura 11.9 declara o operador de negação sobrecarregado (definido na Figura 11.10, linhas 77–80). Esse operador determina se um objeto de nossa classe `String` está vazio. Por exemplo, ao ver a expressão `!string1`, o compilador gera a chamada de função

```
string1.operator!()
```

Essa função simplesmente retorna o resultado de testar se `length` é igual a zero.

Operadores de igualdade e relacionais sobrecarregados

As linhas 23–24 da Figura 11.9 declaram o operador de igualdade sobrecarregado (definido na Figura 11.10, linhas 83–86) e o operador menor que sobrecarregado (definido na Figura 11.10, linhas 89–92) para a classe `String`. Esses são semelhantes, então vamos discutir somente um exemplo, a saber, sobrecarregar o operador `==`. Ao ver a expressão `string1 == string2`, o compilador gera a chamada de função-membro

```
string1.operator==( string2 )
```

que retorna `true` se `string1` for igual a `string2`. Cada um desses operadores utiliza a função `strcmp` (de `<cstring>`) para comparar as strings de caractere nos objetos `String`. Muitos programadores em C++ defendem o uso de algumas funções operadoras sobrecarregadas para implementar outras. Então, os operadores `!=`, `>`, `<=` e `>=` são implementados (Figura 11.9, linhas 27–48) em termos de `operator==` e `operator<`. Por exemplo, a função sobrecarregada `operator>=` (implementada nas linhas 45–48 no arquivo de cabeçalho) utiliza o operador `<` sobrecarregado para determinar se um objeto `String` é maior que ou igual ao outro. Observe que as funções operadoras para `!=`, `>`, `<=` e `>=` são definidas no arquivo de cabeçalho. O compilador coloca essas definições inline para eliminar o overhead das chamadas de função extras.

**Observação de engenharia de software 11.10**

Implementando funções-membro com funções-membro anteriormente definidas, o programador reutiliza o código para reduzir a quantidade de código que deve ser escrita e mantida.

Operadores de subscrito sobrecarregados

As linhas 50–51 no arquivo de cabeçalho declaram dois operadores de subscrito sobrecarregados (definidos na Figura 11.10, linhas 95–106 e 109–120, respectivamente) — um para `Strings não-const` e um para `Strings const`. Ao ver uma expressão como `string1[ 0 ]`, o compilador gera a chamada de função-membro

```
string1.operator[]( 0 )
```

(utilizando a versão apropriada de `operator[]` com base no fato de a `String` ser ou não `const`). Cada implementação de `operator[]` valida primeiro o subscrito para assegurar que ele está no intervalo. Se o subscrito estiver fora do intervalo, cada função imprime uma mensagem de erro e termina o programa com uma chamada a `exit`.⁵ Se o subscrito estiver no intervalo, a versão não-const de `operator[]` retorna um `char &` para o caractere apropriado do objeto `String`; esse `char &` pode ser utilizado como um *lvalue* para modificar o caractere designado do objeto `String`. A versão `const` de `operator[]` retorna o caractere apropriado do objeto `String`; isso pode ser utilizado somente como um *rvalue* para ler o valor do caractere.

**Dica de prevenção de erro 11.2**

É perigoso retornar uma referência não-const `char` de um operador de subscrito sobrecarregado em uma classe `String`. Por exemplo, o cliente poderia utilizar essa referência para inserir um nulo (`'\0'`) em qualquer lugar da string.

Operador de chamada de função sobrecarregado

A linha 52 da Figura 11.9 declara o **operador de chamada de função sobrecarregado** (definido na Figura 11.10, linhas 123–150). Sobrecarregamos esse operador para selecionar uma substring de uma `String`. Os dois parâmetros do tipo inteiro especificam a localização inicial e o comprimento da substring sendo selecionada a partir da `String`. Se a localização inicial estiver fora do intervalo ou o comprimento de substring for negativo, o operador simplesmente retorna uma `String` vazia. Se o comprimento da substring for 0, então a substring é selecionada até o fim do objeto `String`. Por exemplo, suponha `string1` seja um objeto `String` contendo a string "AEIOU". Para a expressão `string1( 2, 2 )`, o compilador gera a chamada de função-membro

⁵ Novamente, é mais apropriado quando um subscrito está fora de intervalo 'lançar uma exceção' indicando o subscrito fora do intervalo.

```
string1.operator()( 2, 2 )
```

Quando essa chamada executa, ela produz um objeto `String` contendo a string "10" e retorna uma cópia desse objeto.

Sobrecarregar o operador de chamada de função `()` é um recurso poderoso, porque as funções podem aceitar listas de parâmetros arbitrariamente longas e complexas. Então podemos utilizar essa capacidade para muitos propósitos interessantes. Esse tipo de utilização do operador de chamada de função é uma notação de subscrito de array alternativa: em vez de utilizar a complicada notação de colchetes duplos para arrays bidimensionais baseados em ponteiro, como em `a[ b ][ c ]`, alguns programadores preferem sobrecarregar o operador de chamada de função para permitir a notação `a( b, c )`. O operador de chamada de função sobrecarregado deve ser uma função-membro não-`static`. Esse operador é utilizado somente quando o 'nome da função' é um objeto da classe `String`.

Função-membro `getLength` da `string`

A linha 53 na Figura 11.9 declara a função `getLength` (definida na Figura 11.10, linhas 153–156), que retorna o comprimento de uma `String`.

Notas sobre nossa classe `String`

Neste ponto, você deve passar pelo código em `main`, examinar a janela de saída e verificar cada utilização de um operador sobrecarregado. Ao estudar a saída, preste atenção especial às chamadas de construtor implícitas que são geradas para criar objetos `String` temporários por todo o programa. Muitas dessas chamadas introduzem um overhead adicional no programa que podem ser evitados se a classe fornecer operadores sobrecarregados que aceitam argumentos `char *`. Entretanto, as funções operadoras adicionais podem tornar a classe mais difícil de manter, modificar e depurar.

11.11 Sobrecarregando ++ e --

As versões prefixada e pós-fixada dos operadores de incremento e decremento podem ser inteiramente sobrecarregadas. Veremos como o compilador distingue entre a versão prefixada e a pós-fixada de um operador de incremento ou decremento.

Para sobrecarregar o operador de incremento para permitir tanto o uso de incremento prefixado como de pós-fixado, toda função operadora sobrecarregada deve ter uma assinatura distinta para que o compilador consiga determinar que versão de ++ é pretendida. As versões prefixadas são sobrecarregadas exatamente como qualquer outro operador unário prefixado seria.

Sobrecarregando o operador de incremento prefixado

Suponha, por exemplo, que quiséssemos adicionar 1 ao dia no objeto `Date d1`. Ao ver a expressão de pré-incremento `++d1`, o compilador gera a chamada de função-membro

```
d1.operator++()
```

O protótipo dessa função operadora seria

```
Date &operator++();
```

Se o operador de incremento prefixado é implementado como uma função global, então, ao ver a expressão `++d1`, o compilador gera a chamada de função

```
operator++( d1 )
```

O protótipo dessa função operadora seria declarado na classe `Date` como

```
Date &operator++( Date & );
```

Sobrecarregando o operador de incremento pós-fixado

Sobrecarregar o operador de incremento pós-fixado apresenta um desafio, porque o compilador deve ser capaz de distinguir entre as assinaturas das funções operadoras sobrecarregadas de incremento prefixado e pós-fixado. A convenção que foi adotada em C++ é que, ao ver a expressão de pós-incremento `d1++`, o compilador gera a chamada de função-membro

```
d1.operator++( 0 )
```

O protótipo dessa função é

```
Date operator++( int )
```

O argumento 0 é estritamente um 'valor fictício' que permite ao compilador distinguir entre as funções operadoras de incremento prefixado e pós-fixado.

Se o incremento pós-fixado é implementado como uma função global, então, quando o compilador vê a expressão `d1++`, ele gera a chamada de função

```
operator++( d1, 0 )
```

O protótipo dessa função seria

```
Date operator++( Date &, int );
```

Mais uma vez, o argumento 0 é utilizado pelo compilador para distinguir entre os operadores de incremento prefixado e pós-fixado implementados como funções globais. Observe que o operador de incremento pós-fixado retorna objetos `Date` por valor, enquanto o

operador de incremento prefixado retorna objetos Date por referência, porque o operador de incremento pós-fixado em geral retorna um objeto temporário que contém o valor original do objeto antes de o incremento ocorrer. O C++ trata esses objetos como *rvalues*, os quais não podem ser utilizados no lado esquerdo de uma atribuição. O operador de incremento prefixado retorna o objeto real incrementado com seu novo valor. Esse objeto pode ser utilizado como um *lvalue* em uma expressão contínua.



Dica de desempenho 11.3

O objeto extra que é criado pelo operador de incremento (ou decremento) pós-fixado pode resultar em um problema de desempenho significativo — especialmente quando o operador é utilizado em um loop. Por essa razão, você só deve utilizar o operador de incremento (ou decremento) pós-fixado quando a lógica do programa exigir pós-incremento (ou pós-decremento).

Tudo declarado nesta seção sobre a sobrecarga de operadores de incremento prefixado e pós-fixado se aplica à sobrecarga de operadores de pré-decremento e pós-decremento. Em seguida, examinamos uma classe Date com operadores de incremento prefixado e pós-fixado sobrecarregados.

11.12 Estudo de caso: uma classe Date

O programa das figuras 11.12–11.14 demonstra uma classe Date. A classe usa operadores de incremento prefixado e pós-fixado sobrecarregados para adicionar 1 ao dia em um objeto Date, enquanto produz incrementos apropriados ao mês e ano, se necessário. O arquivo de cabeçalho Date (Figura 11.12) especifica que a interface `public` de Date inclui um operador de inserção de fluxo sobrecarregado (linha 11), um construtor-padrão (linha 13), uma função `setDate` (linha 14), um operador de incremento prefixado sobrecarregado (linha 15), um operador de incremento pós-fixado sobrecarregado (linha 16), um operador de atribuição de adição `+=` sobrecarregado (linha 17), uma função para testar anos bissextos (linha 18) e uma para determinar se um dia é ou não o último dia do mês (linha 19).

A função `main` (Figura 11.14) cria três objetos Date (linhas 11–13) — `d1` é inicializado por padrão como 1º de janeiro de 1900; `d2` é inicializado como 27 de dezembro de 1992; e `d3` é inicializado como uma data inválida. O construtor Date (definido na Figura 11.13, linhas 11–14) chama `setDate` para validar o mês, dia e ano especificados. Um mês inválido é configurado como 1, um ano inválido é configurado como 1900 e um dia inválido é configurado como 1.

```

1  // Figura 11.12: Date.h
2  // Definição da classe Date.
3  #ifndef DATE_H
4  #define DATE_H
5
6  #include <iostream>
7  using std::ostream;
8
9  class Date
10 {
11     friend ostream &operator<<( ostream &, const Date & );
12 public:
13     Date( int m = 1, int d = 1, int y = 1900 ); // construtor-padrão
14     void setDate( int, int, int ); // configura month, day, year
15     Date &operator++(); // operador de incremento prefixado
16     Date operator++( int ); // operador de incremento pós-fixado
17     const Date &operator+=( int ); // adiciona dias, modifica objeto
18     bool leapYear( int ) const; // a data está em um ano bissexto?
19     bool endOfMonth( int ) const; // a data está no fim do mês?
20 private:
21     int month;
22     int day;
23     int year;
24
25     static const int days[]; // array de dias por mês
26     void helpIncrement(); // função utilitária para incrementar date
27 }; // fim da classe Date
28
29 #endif

```

Figura 11.12 A definição da classe Date com operadores de incremento sobrecarregados.

As linhas 15–16 de `main` geram saída de cada um dos objetos `Date` construídos, utilizando o operador de inserção de fluxo sobrecarregado (definido na Figura 11.13, linhas 96–103). A linha 16 de `main` utiliza o operador `+=` sobrecarregado para adicionar sete dias a `d2`. A linha 18 utiliza a função `setDate` para configurar `d3` como 28 de fevereiro de 1992, que é um ano bissexto. Então, a linha 20 pré-incrementa `d3` para mostrar que a data é incrementada adequadamente como 29 de fevereiro. Em seguida, a linha 22 cria um objeto `Date`, `d4`, que é inicializado com a data 13 de julho de 2002. Depois, a linha 26 incrementa `d4` por 1 com o operador de incremento prefixado sobrecarregado. As linhas 24–27 geram saída de `d4` antes e depois da operação de pré-incremento, para confirmar que ela funcionou corretamente. Por fim, a linha 31 incrementa `d4` com o operador de incremento pós-fixado sobrecarregado. As linhas 29–32 geram saída de `d4` antes e depois da operação de pós-incremento, para confirmar que ela funcionou corretamente.

```

1 // Figura 11.13: Date.cpp
2 // Definições de função-membro da classe Date.
3 #include <iostream>
4 #include "Date.h"
5
6 // inicializa membro static no escopo de arquivo; uma cópia no nível da classe
7 const int Date::days[] =
8     { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
9
10 // construtor Date
11 Date::Date( int m, int d, int y )
12 {
13     setDate( m, d, y );
14 } // fim do construtor Date
15
16 // configura month, day e year
17 void Date::setDate( int mm, int dd, int yy )
18 {
19     month = ( mm >= 1 && mm <= 12 ) ? mm : 1;
20     year = ( yy >= 1900 && yy <= 2100 ) ? yy : 1900;
21
22     // testa se é um ano bissexto
23     if ( month == 2 && leapYear( year ) )
24         day = ( dd >= 1 && dd <= 29 ) ? dd : 1;
25     else
26         day = ( dd >= 1 && dd <= days[ month ] ) ? dd : 1;
27 } // fim da função setDate
28
29 // operador de incremento prefixado sobrecarregado
30 Date &Date::operator++()
31 {
32     helpIncrement(); // incrementa data
33     return *this; // retorno de referência para criar um lvalue
34 } // fim da função operator++
35
36 // operador de incremento pós-fixado sobrecarregado; observe que
37 // parâmetro fictício do tipo inteiro não tem um nome de parâmetro
38 Date Date::operator++( int )
39 {
40     Date temp = *this; // armazena o estado atual do objeto
41     helpIncrement();
42
43     // retorna o objeto temporário, salvo, não incrementado
44     return temp; // retorno de valor; não um retorno de referência
45 } // fim da função operator++
46

```

Figura 11.13 Definições de função-membro e função friend da classe `Date`.

(continua)

```

47 // adiciona número especificado de dias a date
48 const Date &Date::operator+=( int additionalDays )
49 {
50     for ( int i = 0; i < additionalDays; i++ )
51         helpIncrement();
52
53     return *this; // permite cascadeamento
54 } // fim da função operator+=
55
56 // se o ano for um ano bissexto, retorna true; caso contrário, retorna false
57 bool Date::leapYear( int testYear ) const
58 {
59     if ( testYear % 400 == 0 ||
60         ( testYear % 100 != 0 && testYear % 4 == 0 ) )
61         return true; // um ano bissexto
62     else
63         return false; // não um ano bissexto
64 } // fim da função leapYear
65
66 // determina se o dia é o último dia do mês
67 bool Date::endOfMonth( int testDay ) const
68 {
69     if ( month == 2 && leapYear( year ) )
70         return testDay == 29; // último dia de fevereiro em ano bissexto
71     else
72         return testDay == days[ month ];
73 } // fim da função endOfMonth
74
75 // função para ajudar a incrementar a data
76 void Date::helpIncrement()
77 {
78     // dia não é o fim do mês
79     if ( !endOfMonth( day ) )
80         day++; // incrementa dia
81     else
82         if ( month < 12 ) // dia é o fim do mês e mês < 12
83         {
84             month++; // incrementa mês
85             day = 1; // primeiro dia do novo mês
86         } // fim do if
87         else // último dia do ano
88         {
89             year++; // incrementa ano
90             month = 1; // primeiro mês do novo ano
91             day = 1; // primeiro dia do novo mês
92         } // fim do else
93 } // fim da função helpIncrement
94
95 // operador de saída sobrecarregado
96 ostream &operator<<( ostream &output, const Date &d )
97 {
98     static char *monthName[ 13 ] = { "", "January", "February",
99         "March", "April", "May", "June", "July", "August",
100         "September", "October", "November", "December" };
101     output << monthName[ d.month ] << " " << d.day << ", " << d.year;
102     return output; // permite cascadeamento
103 } // fim da função operator<<

```

Figura 11.13 Definições de função-membro e função friend da classe Date.

(continuação)

```

1 // Figura 11.14: fig11_14.cpp
2 // Programa de teste da classe Date.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Date.h" // definição da classe Date
8
9 int main()
10 {
11     Date d1; // assume o padrão de 1º de janeiro de 1900
12     Date d2( 12, 27, 1992 ); // 27 de dezembro de 1992
13     Date d3( 0, 99, 8045 ); // data inválida
14
15     cout << "d1 is " << d1 << "\nd2 is " << d2 << "\nd3 is " << d3;
16     cout << "\n\nd2 += 7 is " << ( d2 += 7 );
17
18     d3.setDate( 2, 28, 1992 );
19     cout << "\n\nd3 is " << d3;
20     cout << "\n++d3 is " << ++d3 << " (leap year allows 29th)";
21
22     Date d4( 7, 13, 2002 );
23
24     cout << "\n\nTesting the prefix increment operator:\n"
25         << " d4 is " << d4 << endl;
26     cout << "++d4 is " << ++d4 << endl;
27     cout << " d4 is " << d4;
28
29     cout << "\n\nTesting the postfix increment operator:\n"
30         << " d4 is " << d4 << endl;
31     cout << "d4++ is " << d4++ << endl;
32     cout << " d4 is " << d4 << endl;
33     return 0;
34 } // fim de main

```

```

d1 is January 1, 1900
d2 is December 27, 1992
d3 is January 1, 1900

d2 += 7 is January 3, 1993

d3 is February 28, 1992
++d3 is February 29, 1992 (leap year allows 29th)

Testing the prefix increment operator:
d4 is July 13, 2002
++d4 is July 14, 2002
d4 is July 14, 2002

Testing the postfix increment operator:
d4 is July 14, 2002
d4++ is July 14, 2002
d4 is July 15, 2002

```

Figura 11.14 Programa de teste da classe Date.

Sobrecarregar o operador de incremento prefixado é simples e direto. O operador de incremento prefixado (definido na Figura 11.13, linhas 30–34) chama a função utilitária `helpIncrement` (definida na Figura 11.13, linhas 76–93) para incrementar a data. Essa função lida com as ‘voltas’ ou ‘reinícios’ que ocorrem quando incrementamos o último dia do mês. Essas voltas requerem incrementar o mês. Se o mês já for 12, então o ano também deve ser incrementado e o mês deve ser configurado como 1. A função `helpIncrement` utiliza a função `endOfMonth` para incrementar o dia corretamente.

O operador de incremento prefixado sobrecarregado retorna uma referência ao objeto `Date` atual (isto é, aquele que acabou de ser incrementado). Isso ocorre porque o objeto atual, `*this`, é retornado como um `Date &`. Isso permite que um objeto `Date` pré-incrementado seja utilizado como um *lvalue*, que é como o operador de incremento prefixado predefinido funciona para tipos fundamentais.

Sobrecarregar o operador de incremento pós-fixado (definido na Figura 11.13, linhas 38–45) é mais difícil. Para emular o efeito do pós-incremento, devemos retornar uma cópia do objeto `Date` não incrementado. Por exemplo, essa variável `int x` tem o valor 7, a instrução

```
cout << x++ << endl;
```

gera saída do valor original da variável `x`. Então, gostaríamos que nosso operador de incremento pós-fixado operasse da mesma maneira que um objeto `Date`. Na entrada para `operator++`, salvamos o objeto atual (`*this`) em `temp` (linha 40). Em seguida, chamamos `helpIncrement` para incrementar o objeto `Date` atual. Então, a linha 44 retorna a cópia não incrementada do objeto previamente armazenado em `temp`. Observe que essa função não pode retornar uma referência ao objeto `Date temp` local, porque a variável local é destruída quando a função em que ela é declarada termina. Portanto, declarar o tipo de retorno para essa função como `Date &` retornaria uma referência a um objeto que não existe mais. Retornar uma referência (ou um ponteiro) a uma variável local é um erro comum para o qual a maioria dos compiladores emitirá um aviso.

11.13 Classe string da biblioteca-padrão

Neste capítulo, você aprendeu que é possível construir uma classe `String` (figuras 11.9–11.11) que é melhor que as strings `char *` no estilo C que C++ absorveu de C. Também aprendeu que pode construir uma classe `Array` (figuras 11.6–11.8) que é melhor que os arrays baseados em ponteiro no estilo C que C++ absorveu de C.

Construir classes úteis reutilizáveis como `String` e `Array` exige trabalho. Entretanto, depois que essas classes foram testadas e depuradas, elas podem ser reutilizadas por você, seus colegas, sua empresa, muitas empresas, uma indústria inteira ou até mesmo muitas indústrias (se forem colocadas em bibliotecas públicas ou comerciais). Os designers de C++ fizeram exatamente isso, construindo a classe `string` (que temos utilizado desde o Capítulo 3) e o template da classe `vector` (que introduzimos no Capítulo 7) no padrão C++. Essas classes estão disponíveis a qualquer pessoa construindo aplicativos com C++. Como você verá no Capítulo 23, “Standard Template Library (STL)”, a C++ Standard Library fornece vários templates de classes predefinidos para você utilizar em programas.

Para fechar este capítulo, refazemos nosso exemplo `String` (figuras 11.9–11.11), utilizando a classe `string` C++ padrão. Reelaboramos nosso exemplo para demonstrar funcionalidades semelhantes fornecidas pela classe `string` padrão. Demonstramos também as três funções-membro da classe `string` padrão — `empty`, `substr` e `at` — que não faziam parte de nossa `String` de exemplo. A função `empty` determina se uma `string` está vazia, a função `substr` retorna uma `string` que representa uma parte de uma `string` existente e a função `at` retorna o caractere em um índice específico de uma `string` (depois de verificar se o índice está no intervalo). O Capítulo 18 apresenta a classe `string` em detalhes.

Classe *string* da biblioteca-padrão

O programa da Figura 11.15 reimplementa o programa da Figura 11.11, utilizando a classe `string` padrão. Como você verá neste exemplo, a classe `string` fornece toda a funcionalidade de nossa classe `String` apresentada nas figuras 11.9–11.10. A classe `string` é definida no cabeçalho `<string>` (linha 7) e pertence ao namespace `std` (linha 8).

As linhas 12–14 criam três objetos `string` — `s1` é inicializado com o literal “happy”, `s2` é inicializado com o literal “birthday” e `s3` utiliza o construtor de `string`-padrão para criar uma `string` vazia. As linhas 17–18 geram saída desses três objetos, utilizando `cout` e o operador `<<`, que os projetistas da classe `string` sobrecarregaram para tratar objetos `string`. Então, as linhas 19–25 mostram os resultados da comparação de `s2` com `s1` utilizando os operadores de igualdade e relacionais sobrecarregados da classe `string`.

```
1 // Figura 11.15: fig11_15.cpp
2 // Programa de teste da classe string da biblioteca-padrão.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string>
8 using std::string;
```

Figura 11.15 Classe `string` da biblioteca-padrão.

(continua)


```

9
10 int main()
11 {
12     string s1( "happy" );
13     string s2( " birthday" );
14     string s3;
15
16     // testa operadores de igualdade e relacionais sobrecarregados
17     cout << "s1 is \" " << s1 << "\"; s2 is \" " << s2
18         << "\"; s3 is \" " << s3 << "\"
19         << "\n\nThe results of comparing s2 and s1:"
20         << "\ns2 == s1 yields " << ( s2 == s1 ? "true" : "false" )
21         << "\ns2 != s1 yields " << ( s2 != s1 ? "true" : "false" )
22         << "\ns2 > s1 yields " << ( s2 > s1 ? "true" : "false" )
23         << "\ns2 < s1 yields " << ( s2 < s1 ? "true" : "false" )
24         << "\ns2 >= s1 yields " << ( s2 >= s1 ? "true" : "false" )
25         << "\ns2 <= s1 yields " << ( s2 <= s1 ? "true" : "false" );
26
27     // testa a função-membro string vazia
28     cout << "\n\nTesting s3.empty(): " << endl;
29
30     if ( s3.empty() )
31     {
32         cout << "s3 is empty; assigning s1 to s3;" << endl;
33         s3 = s1; // atribui s1 a s3
34         cout << "s3 is \" " << s3 << "\";
35     } // fim do if
36
37     // testa operador de concatenação de string sobrecarregado
38     cout << "\n\ns1 += s2 yields s1 = ";
39     s1 += s2; // testa a concatenação sobrecarregada
40     cout << s1;
41
42     // testa operador de concatenação de string sobrecarregado com string no estilo C
43     cout << "\n\ns1 += \" to you\" yields" << endl;
44     s1 += " to you";
45     cout << "s1 = " << s1 << "\n\n";
46
47     // testa função-membro string substr
48     cout << "The substring of s1 starting at location 0 for\n"
49         << "14 characters, s1.substr(0, 14), is:\n"
50         << s1.substr( 0, 14 ) << "\n\n";
51
52     // testa a opção de substr "to-end-of-string"
53     cout << "The substring of s1 starting at\n"
54         << "location 15, s1.substr(15), is:\n"
55         << s1.substr( 15 ) << endl;
56
57     // testa o construtor de cópia
58     string *s4Ptr = new string( s1 );
59     cout << "\n*s4Ptr = " << *s4Ptr << "\n\n";
60
61     // testa o operador de atribuição (=) com a auto-atribuição
62     cout << "assigning *s4Ptr to *s4Ptr" << endl;
63     *s4Ptr = *s4Ptr;
64     cout << "*s4Ptr = " << *s4Ptr << endl;

```

Figura 11.15 Classe string da biblioteca-padrão.

(continua)

```

65
66 // testa o destrutor
67 delete s4Ptr;
68
69 // testa o uso do operador de subscripto para criar lvalue
70 s1[ 0 ] = 'H';
71 s1[ 6 ] = 'B';
72 cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "
73     << s1 << "\n\n";
74
75 // testa o subscripto fora do intervalo com a função-membro string 'at'
76 cout << "Attempt to assign 'd' to s1.at( 30 ) yields:" << endl;
77 s1.at( 30 ) = 'd'; // ERRO: subscripto fora do intervalo
78 return 0;
79 } // fim de main

```

s1 is "happy"; s2 is " birthday"; s3 is ""

The results of comparing s2 and s1:

s2 == s1 yields false

s2 != s1 yields true

s2 > s1 yields false

s2 < s1 yields true

s2 >= s1 yields false

s2 <= s1 yields true

Testing s3.empty():

s3 is empty; assigning s1 to s3;

s3 is "happy"

s1 += s2 yields s1 = happy birthday

s1 += " to you" yields

s1 = happy birthday to you

The substring of s1 starting at location 0 for

14 characters, s1.substr(0, 14), is:

happy birthday

The substring of s1 starting at

location 15, s1.substr(15), is:

to you

*s4Ptr = happy birthday to you

assigning *s4Ptr to *s4Ptr

*s4Ptr = happy birthday to you

s1 after s1[0] = 'H' and s1[6] = 'B' is: Happy Birthday to you

Attempt to assign 'd' to s1.at(30) yields:

abnormal program termination

Figura 11.15 Classe string da biblioteca-padrão.

(continuação)

Nossa classe `String` (figuras 11.9–11.10) forneceu um operador! sobrecarregado que testou uma `String` para determinar se ela estava vazia ou não. A classe `string` padrão não fornece essa funcionalidade como um operador sobrecarregado; em vez disso, fornece a função-membro `empty`, que demonstramos na linha 30. A função-membro `empty` retorna `true` se a `string` estiver vazia; caso contrário, retorna `false`.

A linha 33 demonstra o operador de atribuição sobrecarregado da classe `string` atribuindo `s1` a `s3`. A linha 34 gera saída de `s3` para demonstrar que a atribuição funcionou corretamente.

A linha 39 demonstra o operador `+=` sobrecarregado da classe `string` para concatenação de `string`. Nesse caso, o conteúdo de `s2` é acrescentado a `s1`. Então a linha 40 gera a saída da `string` resultante que é armazenada em `s1`. A linha 44 demonstra que um literal `string` no estilo C pode ser acrescentado a um objeto `string` utilizando operador `+=`. A linha 45 exibe o resultado.

Nossa classe `String` (figuras 11.9–11.10) forneceu o operador() sobrecarregado para obter substrings. A classe `string` padrão não fornece essa funcionalidade como um operador sobrecarregado; em vez disso, fornece a função-membro `substr` (linhas 50 e 55). A chamada a `substr` na linha 50 obtém uma substring de 14 caracteres (especificada pelo segundo argumento) de `s1` iniciando na posição 0 (especificada pelo primeiro argumento). A chamada a `substr` na linha 55 obtém uma substring que inicia da posição 15 de `s1`. Quando o segundo argumento não é especificado, `substr` retorna o restante da `string` em que ela é chamada.

A linha 58 aloca um objeto `string` dinamicamente e inicializa com uma cópia de `s1`. Isso resulta em uma chamada para o construtor de cópia da classe `string`. A linha 63 utiliza o operador `=` sobrecarregado da classe `string` para demonstrar que ela trata a auto-atribuição adequadamente.

As linhas 70–71 utilizaram o operador `[]` sobrecarregado da classe `string` para criar *lvalues* que permitem que novos caracteres substituam os caracteres existentes em `s1`. A linha 73 gera saída do novo valor de `s1`. Em nossa classe `String` (figuras 11.9–11.10), o operador `[]` sobrecarregado realizou verificação de limites para determinar se o subscrito que ele recebeu como um argumento era um subscrito válido na `string`. Se o subscrito fosse inválido, o operador imprimiria uma mensagem de erro e terminaria o programa. O operador `[]` sobrecarregado da classe `string` padrão não realiza nenhuma verificação de limites. Portanto, o programador deve assegurar que as operações que utilizam o operador `[]` sobrecarregado da classe `string` padrão não manipulam acidentalmente os elementos fora dos limites da `string`. A classe `string` padrão fornece verificação de limites em sua função-membro `at`, que ‘lança uma exceção’ se seu argumento for um subscrito inválido. Por padrão, isso termina um programa C++⁶. Se o subscrito for válido, a função `at` retorna o caractere na localização especificada como um *lvalue* modificável ou um *lvalue* não modificável (isto é, uma referência `const`), dependendo do contexto em que a chamada aparece. A linha 77 demonstra uma chamada à função `at` com um subscrito inválido.

11.14 Construtores explicit

Nas seções 11.8 e 11.9, discutimos que qualquer construtor de um único argumento pode ser utilizado pelo compilador para realizar uma conversão implícita — o tipo recebido pelo construtor é convertido em um objeto da classe em que o construtor é definido. A conversão é automática e o programador não precisa utilizar um operador de coerção. Em algumas situações, as conversões implícitas são indesejáveis ou propensas a erros. Por exemplo, nossa classe `Array` na Figura 11.6 define um construtor que aceita um único argumento `int`. A intenção desse construtor é criar um objeto `Array` contendo o número de elementos especificado pelo argumento `int`. Entretanto, esse construtor pode ser mal-empregado pelo compilador para realizar uma conversão implícita.



Erro comum de programação 11.9

Infelizmente, o compilador poderia utilizar conversões implícitas em casos que você não espera, resultando em expressões ambíguas que geram erros de compilação ou resultam em erros de lógica em tempo de execução.

Utilizando acidentalmente um construtor de um único argumento como um construtor de conversão

O programa (Figura 11.16) utiliza a classe `Array` das figuras 11.6–11.7 para demonstrar uma conversão implícita inadequada.

A linha 13 em `main` instancia o objeto `Array integers1` e chama o construtor de um único argumento com o valor `int 7` para especificar o número de elementos no `Array`. Lembre-se, a partir do que foi visto na Figura 11.7, de que o construtor `Array` que recebe um argumento `int` inicializa todos os elementos do array como 0. A linha 14 chama a função `outputArray` (definida nas linhas 20–24), que recebe como seu argumento um `const Array &` para um `Array`. A função gera saída do número de elementos em seu argumento `Array` e do conteúdo do `Array`. Nesse caso, o tamanho do `Array` é 7, então sete 0s são enviados para a saída.

A linha 15 chama a função `outputArray` com o valor `int 3` como um argumento. Entretanto, esse programa não contém uma função chamada `outputArray` que aceita um argumento `int`. Então, o compilador determina se a classe `Array` fornece um construtor de conversão que pode converter um `int` em um `Array`. Como qualquer construtor que recebe um argumento único é considerado um construtor de conversão, o compilador assume que o construtor `Array` que recebe um único `int` é um construtor de conversão e o utiliza para converter o argumento 3 em um objeto `Array` temporário que contém três elementos. Então, o compilador passa o objeto `Array` temporário à função `outputArray` para gerar a saída do conteúdo do `Array`. Portanto, mesmo que não forneçamos explicitamente uma função `outputArray` que recebe um argumento `int`, o compilador consegue compilar a linha 15. A saída mostra o conteúdo do `Array` de três elementos contendo 0s.

⁶ Novamente, o Capítulo 16, “Tratamento de exceções” demonstra como ‘capturar’ essas exceções.

```

1 // Figura 11.16: fig11_16.cpp
2 // Driver para a classe Array simples.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Array.h"
8
9 void outputArray( const Array & ); // protótipo
10
11 int main()
12 {
13     Array integers1( 7 ); // array de 7 elementos
14     outputArray( integers1 ); // gera saída do Array integers1
15     outputArray( 3 ); // converte 3 em um Array e gera saída do conteúdo de Array
16     return 0;
17 } // fim de main
18
19 // imprime conteúdo de Array
20 void outputArray( const Array &arrayToOutput )
21 {
22     cout << "The Array received has " << arrayToOutput.getSize()
23         << " elements. The contents are:\n" << arrayToOutput << endl;
24 } // fim de outputArray

```

The Array received has 7 elements. The contents are:

```

0      0      0      0
0      0      0

```

The Array received has 3 elements. The contents are:

```

0      0      0

```

Figura 11.16 Construtores de um único argumento e conversões implícitas.

Impedindo a utilização acidental de um construtor de um único argumento como um construtor de conversão

O C++ fornece a palavra-chave **explicit** para suprimir as conversões implícitas via construtores de conversão quando essas conversões não devem ser permitidas. Um construtor que é declarado **explicit** não pode ser utilizado em uma conversão implícita. A Figura 11.17 declara um construtor **explicit** na classe `Array`. A única modificação no `Array.h` foi a adição da palavra-chave **explicit** à declaração do construtor de um único argumento na linha 15. Nenhuma modificação é requerida no arquivo de código-fonte contendo definições de função-membro da classe `Array`.

A Figura 11.18 apresenta uma versão ligeiramente modificada do programa na Figura 11.16. Quando esse programa é compilado, o compilador produz uma mensagem de erro que indica que o valor do tipo inteiro passado para `outputArray` na linha 15 não pode ser convertido em um `const Array &`. A mensagem de erro de compilador é mostrada na janela de saída. A linha 16 demonstra como o construtor explícito pode ser utilizado para criar um `Array` temporário de 3 elementos e passá-lo para a função `outputArray`.



Erro comum de programação 11.10

Tentar invocar um construtor `explicit` para uma conversão implícita é um erro de compilação.



Erro comum de programação 11.11

Utilizar a palavra-chave `explicit` em membros de dados ou funções-membro diferentes de um construtor de um único argumento é um erro de compilação.

```

1 // Figura 11.17: Array.h
2 // Classe Array para armazenar arrays de inteiros.
3 #ifndef ARRAY_H
4 #define ARRAY_H
5
6 #include <iostream>
7 using std::ostream;
8 using std::istream;
9
10 class Array
11 {
12     friend ostream &operator<<( ostream &, const Array & );
13     friend istream &operator>>( istream &, Array & );
14 public:
15     explicit Array( int = 10 ); // construtor-padrão
16     Array( const Array & ); // construtor de cópia
17     ~Array(); // destrutor
18     int getSize() const; // retorna tamanho
19
20     const Array &operator=( const Array & ); // operador de atribuição
21     bool operator==( const Array & ) const; // operador de igualdade
22
23     // operador de desigualdade; retorna o oposto do operador ==
24     bool operator!=( const Array &right ) const
25     {
26         return ! ( *this == right ); // invoca Array::operator==
27     } // fim da função operator!=
28
29     // operador de subscrito para objetos não-const retorna lvalue
30     int &operator[]( int );
31
32     // operador de subscrito de objetos const retorna rvalue
33     const int &operator[]( int ) const;
34 private:
35     int size; // tamanho do array baseado em ponteiro
36     int *ptr; // ponteiro para o primeiro elemento do array baseado em ponteiro
37 }; // fim da classe Array
38
39 #endif

```

Figura 11.17 Definição da classe Array com o construtor explicit.

```

1 // Figura 11.18: fig11_18.cpp
2 // Driver para a classe Array simples.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Array.h"
8
9 void outputArray( const Array & ); // protótipo
10
11 int main()
12 {

```

Figura 11.18 Demonstrando um construtor explicit.

(continua)

```

13     Array integers1( 7 ); // array de 7 elementos
14     outputArray( integers1 ); // gera saída do Array integers1
15     outputArray( 3 ); // converte 3 em um Array e gera saída do conteúdo do Array
16     outputArray( Array( 3 ) ); // chamada do construtor explicit de um único argumento
17     return 0;
18 } // fim de main
19
20 // imprime o conteúdo do array
21 void outputArray( const Array &arrayToOutput )
22 {
23     cout << "The Array received has " << arrayToOutput.getSize()
24         << " elements. The contents are:\n" << arrayToOutput << endl;
25 } // fim de outputArray

```

```

c:\cpphttp5_examples\ch11\Fig11_17_18\Fig11_18.cpp(15) : error C2664:
'outputArray' : cannot convert parameter 1 from 'int' to 'const Array &'
Reason: cannot convert from 'int' to 'const Array'
Constructor for class 'Array' is declared 'explicit'

```

Figura 11.18 Demonstrando um construtor explicit.

(continuação)



Dica de prevenção de erro 11.3

Utilize a palavra-chave *explicit* em construtores de um único argumento que não devem ser utilizados pelo compilador para realizar conversões implícitas.

11.15 Síntese

Neste capítulo, você aprendeu a construir classes mais robustas definindo operadores sobrecarregados que permitem aos programadores tratar objetos de suas classes como se eles fossem tipos de dados fundamentais do C++. Apresentamos os conceitos básicos da sobrecarga de operadores, bem como várias restrições que o C++ padrão impõe aos operadores sobrecarregados. Você aprendeu as razões de implementar operadores sobrecarregados como funções-membro ou funções globais. Discutimos as diferenças entre sobrecarregar operadores unários e binários como funções-membro e funções globais. Com as funções globais, mostramos como permitir entrada e saída de objetos de nossas classes utilizando os operadores de extração e de inserção de fluxo sobrecarregados, respectivamente. Mostramos uma sintaxe especial que é requerida para diferenciar entre as versões prefixada e pós-fixada do operador de incremento (++). Demonstramos também a classe string C++ padrão, que faz extenso uso de operadores sobrecarregados para criar uma classe reutilizável robusta que pode substituir strings baseadas em ponteiro no estilo C. Por fim, você aprendeu a utilizar a palavra-chave *explicit* para impedir o compilador de utilizar um construtor de um único argumento para realizar conversões implícitas. No próximo capítulo, continuamos nossa discussão sobre classes introduzindo uma forma de reutilização de software chamada herança. Veremos que as classes frequentemente compartilham atributos e comportamentos comuns. Nesses casos, é possível definir os atributos e comportamentos em uma classe básica comum e ‘herdar’ essas capacidades em novas definições de classe.

Resumo

- O C++ permite ao programador sobrecarregar a maioria dos operadores para se tornarem sensíveis ao contexto em que são utilizados — o compilador gera o código apropriado com base no contexto (em particular, os tipos dos operandos).
- Muitos operadores do C++ podem ser sobrecarregados para trabalhar com tipos definidos pelo usuário.
- Um exemplo de um operador sobrecarregado construído em C++ é o operador <<, que é utilizado como o operador de inserção de fluxo e como o operador de deslocamento de bits para a esquerda. De maneira semelhante, >> também é sobrecarregado; é utilizado como o operador de extração de fluxo e como o operador de bits de deslocamento para a direita. Esses dois operadores são sobrecarregados na C++ Standard Library.
- A própria linguagem C++ sobrecarrega + e -. Esses operadores executam de modo diferente, dependendo de seu contexto na aritmética de inteiros, na aritmética de ponto flutuante e na aritmética de ponteiros.
- Os trabalhos realizados por operadores sobrecarregados também podem ser realizados por chamadas de função, mas a notação de operador é frequentemente mais clara e mais familiar para programadores.

- Um operador é sobrecarregado escrevendo uma definição de função-membro não-`static` ou definição de função global em que o nome da função é a palavra-chave `operator` seguida pelo símbolo do operador sendo sobrecarregado.
- Quando operadores são sobrecarregados como funções-membro, eles devem ser não-`static`, porque devem ser chamados sobre um objeto da classe e operar sobre esse objeto.
- Para utilizar um operador em objetos de classe, esse operador deve ser sobrecarregado — com três exceções: o operador de atribuição (`=`), o operador de endereço (`&`) e o operador vírgula (`,`).
- Você não pode mudar a precedência e a associatividade de um operador sobrecarregando-o.
- Você não pode mudar a 'aridade' de um operador (isto é, o número de operandos que um operador aceita).
- Você não pode criar novos operadores; somente os operadores existentes podem ser sobrecarregados.
- Você não pode mudar o significado de como um operador funciona em objetos de tipos fundamentais.
- Sobrecarregar um operador de atribuição e um operador de adição para uma classe não implica que o operador `+=` também é sobrecarregado. Esse comportamento só pode ser alcançado sobrecarregando explicitamente o operador `+=` dessa classe.
- As funções operadoras podem ser funções-membros ou globais; as funções globais são frequentemente feitas `friends` por razões de desempenho. As funções-membro utilizam o ponteiro `this` implicitamente para obter um de seus argumentos de objeto de classe (o operando esquerdo para operadores binários). Os argumentos para ambos os operandos de um operador binário devem ser explicitamente listados em uma chamada de função global.
- Ao sobrecarregar `()`, `[]`, `->` ou qualquer um dos operadores de atribuição, a função de sobrecarga de operadores deve ser declarada como membros de classe. Para os outros operadores, as funções de sobrecarga de operadores podem ser membros de classe ou funções globais.
- Quando uma função operadora é implementada como uma função-membro, o operando na extrema esquerda (ou único) deve ser um objeto (ou uma referência a um objeto) da classe do operador.
- Se o operando esquerdo precisar ser um objeto de uma classe diferente ou um tipo fundamental, essa função operadora deve ser implementada como uma função global.
- Uma função operadora global pode se tornar um `friend` de uma classe se essa função precisar acessar membros `private` ou `protected` dessa classe diretamente.
- O operador de inserção de fluxo (`<<`) sobrecarregado é utilizado em uma expressão em que o operando esquerdo tem o tipo `ostream &`. Por essa razão, ele deve ser sobrecarregado como uma função global. Para ser uma função-membro, o operador `<<` teria de ser membro da classe `ostream`, mas isso não é possível, uma vez que não temos permissão para modificar classes da C++ Standard Library. De maneira semelhante, o operador de extração de fluxo (`>>`) sobrecarregado deve ser uma função global.
- Outra razão para escolher uma função global para sobrecarregar um operador é permitir que o operador seja comutativo.
- Quando utilizado com `cin` e `strings`, `setw` restringe o número de caracteres lidos ao número de caracteres especificados por seu argumento.
- A função-membro `istream::ignore` descarta o número especificado de caracteres no fluxo de entrada (um caractere por padrão).
- Os operadores de entrada e saída sobrecarregados são declarados como `friends` se precisarem acessar membros de classe não-`public` diretamente por razões de desempenho.
- Um operador unário de uma classe pode ser sobrecarregado como uma função-membro não-`static` sem argumentos ou como uma função global com um argumento; esse argumento deve ser um objeto da classe ou uma referência a um objeto da classe.
- As funções-membro que implementam os operadores sobrecarregados devem ser não-`static` para que possam acessar os dados não-`static` em cada objeto da classe.
- Um operador binário é sobrecarregado como uma função-membro não-`static` com um argumento ou como uma função global com dois argumentos (um desses argumentos deve ser um objeto de classe ou uma referência a um objeto de classe).
- Um construtor de cópia inicializa um novo objeto de uma classe copiando os membros de um objeto existente dessa classe. Quando os objetos de uma classe contêm memória dinamicamente alocada, a classe deve fornecer um construtor de cópia para assegurar que cada cópia de um objeto tem sua própria cópia separada da memória dinamicamente alocada. Em geral, essa classe também forneceria um destrutor e um operador de atribuição sobrecarregado.
- A implementação da função-membro `operator=` deve testar a auto-atribuição, em que um objeto está sendo atribuído a ele mesmo.
- O compilador chama a versão `const` de `operator[]` quando o operador de subscrito é utilizado em um objeto `const` e chama a versão não-`const` do operador quando ele é utilizado em um objeto não-`const`.
- O operador de subscrito de array (`[]`) não está restrito ao uso com arrays. Ele pode ser utilizado para selecionar elementos de outros tipos de classes contêineres. Além disso, com a sobrecarga, os valores de índice não precisam mais ser do tipo inteiro; os caracteres ou strings poderiam ser utilizados, por exemplo.
- O compilador não pode saber antecipadamente como converter entre tipos definidos pelo usuário, e entre tipos fundamentais, portanto o programador deve especificar como fazer isso. Essas conversões podem ser realizadas com construtores de conversão — os construtores de um único argumento que transformam objetos de outros tipos (incluindo tipos fundamentais) em objetos de uma classe particular.

- Um operador de conversão (também chamado de operador de coerção) pode ser utilizado para converter um objeto de uma classe em um objeto de outra classe ou em um objeto de um tipo fundamental. Esse operador de conversão deve ser uma função-membro não-`static`. Funções operadoras de coerção sobrecarregadas podem ser definidas para converter objetos de tipos definidos pelo usuário em tipos fundamentais ou em objetos de outros tipos definidos pelo usuário.
- Uma função operadora de coerção sobrecarregada não especifica um tipo de retorno — o tipo de retorno é o tipo no qual o objeto está sendo convertido.
- Uma das características interessantes dos operadores de coerção e construtores de conversão é que, quando necessário, o compilador pode chamar essas funções implicitamente para criar objetos temporários.
- Qualquer construtor de um único argumento pode ser considerado um construtor de conversão.
- Sobrecarregar o operador de chamada de função `()` é um recurso poderoso, porque as funções podem aceitar listas de parâmetros arbitrariamente longas e complexas.
- Os operadores de incremento e de decremento prefixado e pós-fixado podem ser inteiramente sobrecarregados.
- Para sobrecarregar o operador de incremento para permitir tanto o uso de pré-incremento como de pós-incremento, cada função operadora sobrecarregada deve ter uma assinatura distinta, de modo que o compilador seja capaz de determinar que versão de `++` é pretendida. As versões prefixadas são sobrecarregadas exatamente como qualquer outro operador unário prefixado seria. Fornecer uma assinatura única para o operador de incremento pós-fixado é realizado fornecendo um segundo argumento, que deve ser do tipo `int`. Esse argumento não é fornecido no código-cliente. Ele é utilizado implicitamente pelo compilador para distinguir entre as versões de prefixado e pós-fixado do operador de incremento.
- A classe `string` padrão é definida no cabeçalho `<string>` e pertence ao namespace `std`.
- A classe `string` fornece muitos operadores sobrecarregados, incluindo operadores de igualdade, relacionais, de atribuição, de atribuição de adição (para concatenação) e de subscrito.
- A classe `string` fornece a função-membro `empty`, que retorna `true` se a `string` estiver vazia; caso contrário, retorna `false`.
- A função-membro `substr` da classe `string` obtém uma substring de um comprimento especificado pelo segundo argumento, iniciando na posição especificada pelo primeiro argumento. Quando o segundo argumento não é especificado, `substr` retorna o restante da `string` em que ela é chamada.
- O operador `[]` sobrecarregado da classe `string` não realiza nenhuma verificação de limites. Portanto, o programador deve assegurar que as operações que utilizam o operador `[]` sobrecarregado da classe `string` padrão não manipulam acidentalmente os elementos fora dos limites da `string`.
- A classe `string` padrão fornece a verificação de limites com função-membro `at`, que ‘lança uma exceção’ se seu argumento for um subscrito inválido. Por padrão, isso faz com que um programa C++ termine. Se o subscrito for válido, a função `at` retornará o caractere na localização especificada como um *lvalue* ou um *rvalue*, dependendo do contexto em que a chamada aparecer.
- O C++ fornece a palavra-chave `explicit` para suprimir conversões implícitas via construtores de conversão quando essas conversões não puderem ser permitidas. Um construtor que é declarado `explicit` não pode ser utilizado em uma conversão implícita.

Terminologia

‘aridade’ de um operador	operador <code>!=</code> sobrecarregado	operadores sobrecarregáveis
Array, classe	operador <code>()</code> sobrecarregado	operador <code>!</code>
associatividade não alterada pela sobrecarga	operador <code>[]</code> sobrecarregado	operador <code>!=</code>
auto-atribuição	operador <code>+</code> sobrecarregado	operador <code>()</code>
construtor de conversão	operador <code>++</code> sobrecarregado	operador, palavra-chave
construtor de cópia	operador <code>++(int)</code> sobrecarregado	operador <code>[]</code>
conversão definida pelo usuário	operador <code>+=</code> sobrecarregado	operador <code>+</code>
conversão entre tipos fundamentais e tipos de classe	operador <code><</code> sobrecarregado	operador <code>++</code>
conversões implícitas definidas pelo usuário	operador <code><<</code> sobrecarregado	operador <code>++(int)</code>
<code>empty</code> , função-membro de <code>string</code>	operador <code><=</code> sobrecarregado	operador <code><</code>
<code>explicit</code> , construtor	operador <code>==</code> sobrecarregado	operador <code><<</code>
função global para sobrecarregar um operador	operador <code>></code> sobrecarregado	operador <code>=</code>
função operadora	operador <code>>=</code> sobrecarregado	operador <code>==</code>
função operadora de coerção	operador <code>>></code> sobrecarregado	operador <code>>=</code>
funções operadoras de atribuição	operador de atribuição <code>(=)</code> sobrecarregado	operador <code>>></code>
<code>ignore</code> , função-membro de <code>istream</code>	operador de chamada de função <code>()</code>	sobrecarga de operadores
<i>lvalue</i> (‘left value’)	operador de conversão	sobrecarregando um operador binário
operação de comutatividade	operadores de inserção e extração de fluxo	sobrecarregando um operador unário
operador <code>!</code> sobrecarregado	sobrecarregados	<code>string</code> (classe C++ padrão)

string, concatenação
 substr, função-membro de string

substring
 tipo definido pelo usuário

versão const de operator[]

Exercícios de revisão

- 11.1** Preencha as lacunas em cada uma das seguintes sentenças:
- Suponha que *a* e *b* sejam variáveis do tipo inteiro e que formulamos a soma *a + b*. Agora suponha que *c* e *d* sejam variáveis de ponto flutuante e formulamos a soma *c + d*. Os dois operadores *+* aqui estão sendo claramente utilizados para propósitos diferentes. Esse é um exemplo de _____.
 - A palavra-chave _____ introduz uma definição de função de operador sobrecarregado.
 - Para utilizar os operadores em objetos de classe, eles devem ser sobrecarregados, com exceção dos operadores de _____, _____ e _____.
 - A _____, a _____ e a _____ de um operador não podem ser alteradas sobrecarregando o operador.
- 11.2** Explique os múltiplos significados dos operadores *<<* e *>>* em C++.
- 11.3** Em que contexto o nome *operator* poderia ser utilizado em C++?
- 11.4** (Verdadeiro/falso) Em C++, apenas os operadores existentes podem ser sobrecarregados.
- 11.5** No que a precedência de um operador sobrecarregado em C++ é comparável com a precedência do operador original?

Respostas dos exercícios de revisão

- 11.1** a) sobrecarga de operadores. b) *operator*. c) atribuição (=), endereço (&), vírgula (,). d) precedência, associatividade, 'aridade'.
- 11.2** O operador *>>* é tanto o operador de deslocamento para a direita como o operador de extração de fluxo, dependendo de seu contexto. O operador *<<* é tanto o operador de deslocamento para a esquerda como o operador de inserção de fluxo, dependendo de seu contexto.
- 11.3** Para a sobrecarga de operadores: seria o nome de uma função que forneceria uma versão sobrecarregada do operador / para uma classe específica.
- 11.4** Verdadeiro.
- 11.5** A precedência é idêntica.

Exercícios

- 11.6** Forneça quantos exemplos puder de sobrecarga de operadores implícita em C++. Dê um exemplo razoável de uma situação em que você poderia querer sobrecarregar explicitamente um operador em C++.
- 11.7** Os operadores que não podem ser sobrecarregados são _____, _____, _____ e _____.
- 11.8** A concatenação de string requer dois operandos — as duas strings que devem ser concatenadas. No texto, mostramos como implementar um operador de concatenação sobrecarregado que concatena o segundo objeto *String* com o lado direito do primeiro objeto *String*, modificando, assim, o primeiro objeto *String*. Em alguns aplicativos, é desejável produzir um objeto *String* concatenado sem modificar os argumentos *String*. Implemente o *operator+* para permitir operações como
- ```
string1 = string2 + string3;
```
- 11.9** (*Exercício final de sobrecarga de operadores*) Para apreciar o cuidado que deve haver na seleção de operadores para sobrecarga, liste cada um dos operadores sobrecarregáveis do C++ e, para cada um deles, liste um possível significado (ou vários, se apropriado) para cada uma de várias classes que você estudou neste texto. Sugerimos que você tente:
- Array
  - Stack
  - String
- Depois de fazer isso, comente quais operadores parecem ter significado para uma ampla variedade de classes. Que operadores parecem ser de pouco valor para a sobrecarga? Que operadores parecem ambíguos?
- 11.10** Agora trabalhe no processo descrito no Exercício 11.9 na ordem inversa. Liste cada um dos operadores sobrecarregáveis do C++. Para cada um, liste o que você acredita que poderia ser a 'melhor operação' que o operador deve ser utilizado para representar. Se houver várias operações excelentes, liste todas elas.
- 11.11** Um exemplo interessante de como sobrecarregar o operador de chamada de função (*()*) é permitir outra forma popular de subscrito de array duplo em algumas linguagens de programação. Em vez de dizer
- ```
chessBoard[ row ][ column ]
```
- para um array de objetos, sobrecarregue o operador de chamada de função para permitir a forma alternativa

```
chessBoard( row, column )
```

Crie uma classe `DoubleSubscriptedArray` que tem recursos semelhantes à classe `Array` nas figuras 11.6–11.7. Em tempo de construção, a classe deve ser capaz de criar um array com qualquer número de linhas e qualquer número de colunas. A classe deve fornecer o operador `()` para realizar as operações de subscrito duplo. Por exemplo, em um `DoubleSubscriptedArray` 3 por 5 chamado `a`, o usuário poderia escrever `a( 1, 3 )` para acessar o elemento na linha 1 e a coluna 3. Lembre-se de que o operador `()` pode receber qualquer número de argumentos (veja a classe `String` nas figuras 11.9–11.10 para obter um exemplo do operador `()`). A representação subjacente do array de subscrito duplo deve ser um array de um único subscrito de inteiros com o número de `rows * columns` de elementos. A função `operator()` deve realizar a aritmética de ponteiro adequada para acessar cada elemento do array. Há duas versões de `operator()` — uma que retorna `int &` (de modo que um elemento de um `DoubleSubscriptedArray` possa ser utilizado como um *lvalue*) e uma que retorna `const int &` (de modo que um elemento de um `const DoubleSubscriptedArray` possa ser utilizado somente como um *rvalue*). A classe também deve fornecer os seguintes operadores: `==`, `!=`, `=`, `<<` (para gerar saída do array no formato tabular) e `>>` (para inserir o conteúdo inteiro do array).

11.12 Sobrecarregue o operador de subscrito para retornar o maior elemento de uma coleção, o segundo maior, o terceiro maior e assim por diante.

11.13 Considere a classe `Complex` mostrada nas figuras 11.19–11.21. A classe permite operações nos chamados *números complexos*. Esses são números na forma `parteReal + parteImaginária * i`, onde *i* tem o valor

$$\sqrt{-1}$$

- a) Modifique a classe para permitir a entrada e saída de números complexos pelos operadores sobrecarregados `>>` e `<<`, respectivamente (você deve remover a função `print` da classe).
- b) Sobrecarregue o operador de multiplicação para permitir multiplicação de dois números complexos como em álgebra.
- c) Sobrecarregue os operadores `==` e `!=` para permitir comparações de números complexos.

11.14 Uma máquina com inteiros de 32 bits pode representar inteiros no intervalo de aproximadamente –2 bilhões a +2 bilhões. Essa restrição de tamanho fixo raramente causa problemas, mas há aplicativos em que gostaríamos de ser capazes de utilizar um espectro mais amplo de inteiros. É para isso que o C++ foi construído, a saber, para criar novos tipos de dados poderosos. Considere a classe `HugeInt` das figuras 11.22–11.24. Estude a classe cuidadosamente, então resolva as seguintes questões:

- a) Descreva precisamente como ela opera.
- b) Que restrições a classe tem?
- c) Sobrecarregue o operador de multiplicação `*`.
- d) Sobrecarregue o operador de divisão `/`.
- e) Sobrecarregue todos os operadores de igualdade e relacionais.

[Nota: Não mostramos um operador de atribuição nem um construtor de cópia para a classe `HugeInteger`, porque o operador de atribuição e o construtor de cópia fornecidos pelo compilador são capazes de copiar todo o membro de dados do array adequadamente.]

```
1 // Figura 11.19: Complex.h
2 // Definição da classe Complex.
3 #ifndef COMPLEX_H
4 #define COMPLEX_H
5
6 class Complex
7 {
8 public:
9     Complex( double = 0.0, double = 0.0 ); // construtor
10     Complex operator+( const Complex & ) const; // adição
11     Complex operator-( const Complex & ) const; // subtração
12     void print() const; // saída
13 private:
14     double real; // parte real
15     double imaginary; // parte imaginária
16 }; // fim da classe Complex
17
18 #endif
```

Figura 11.19 Definição da classe `Complex`.

```

1 // Figura 11.20: Complex.cpp
2 // Definições de função-membro da classe Complex.
3 #include <iostream>
4 using std::cout;
5
6 #include "Complex.h" // definição da classe Complex
7
8 // Construtor
9 Complex::Complex( double realPart, double imaginaryPart )
10 : real( realPart ),
11   imaginary( imaginaryPart )
12 {
13     // corpo vazio
14 } // fim do construtor Complex
15
16 // operador de adição
17 Complex Complex::operator+( const Complex &operand2 ) const
18 {
19     return Complex( real + operand2.real,
20                    imaginary + operand2.imaginary );
21 } // fim da função operator+
22
23 // operador de subtração
24 Complex Complex::operator-( const Complex &operand2 ) const
25 {
26     return Complex( real - operand2.real,
27                    imaginary - operand2.imaginary );
28 } // fim da função operator-
29
30 // exibe um objeto Complex na fórmula: (a, b)
31 void Complex::print() const
32 {
33     cout << '(' << real << ", " << imaginary << ')';
34 } // fim da função print

```

Figura 11.20 Definições de função-membro da classe Complex.

```

1 // Figura 11.21: fig11_21.cpp
2 // Programa de teste da classe Complex.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Complex.h"
8
9 int main()
10 {
11     Complex x;
12     Complex y( 4.3, 8.2 );
13     Complex z( 3.3, 1.1 );
14
15     cout << "x: ";
16     x.print();

```

Figura 11.21 Números complexos.

(continua)

```

17     cout << "\ny: ";
18     y.print();
19     cout << "\nz: ";
20     z.print();
21
22     x = y + z;
23     cout << "\n\nx = y + z:" << endl;
24     x.print();
25     cout << " = ";
26     y.print();
27     cout << " + ";
28     z.print();
29
30     x = y - z;
31     cout << "\n\nx = y - z:" << endl;
32     x.print();
33     cout << " = ";
34     y.print();
35     cout << " - ";
36     z.print();
37     cout << endl;
38     return 0;
39 } // fim de main

```

x: (0, 0)

y: (4.3, 8.2)

z: (3.3, 1.1)

x = y + z:

(7.6, 9.3) = (4.3, 8.2) + (3.3, 1.1)

x = y - z:

(1, 7.1) = (4.3, 8.2) - (3.3, 1.1)

Figura 11.21 Números complexos.

(continuação)

```

1 // Figura 11.22: Hugeint.h
2 // Definição da classe HugeInt.
3 #ifndef HUGEINT_H
4 #define HUGEINT_H
5
6 #include <iostream>
7 using std::ostream;
8
9 class HugeInt
10 {
11     friend ostream &operator<<( ostream &, const HugeInt & );
12 public:
13     HugeInt( long = 0 ); // construtor de conversão/padrão
14     HugeInt( const char * ); // construtor de conversão
15
16     // operador de adição; HugeInt + HugeInt
17     HugeInt operator+( const HugeInt & ) const;

```

Figura 11.22 Definição da classe HugeInt.

(continua)

```

18
19 // operador de adição; HugeInt + int
20 HugeInt operator+( int ) const;
21
22 // operador de adição;
23 // HugeInt + string que representa o valor de tipo inteiro grande
24 HugeInt operator+( const char * ) const;
25 private:
26     short integer[ 30 ];
27 }; // fim da classe HugeInt
28
29 #endif

```

Figura 11.22 Definição da classe HugeInt.

(continuação)

```

1 // Figura 11.23: Hugeint.cpp
2 // Definições de função-membro e função friend HugeInt.
3 #include <cctype> // protótipo de função isdigit
4 #include <cstring> // protótipo de função strlen
5 #include "Hugeint.h" // definição da classe HugeInt
6
7 // construtor-padrão; construtor de conversão que
8 // converte um inteiro long em um objeto HugeInt
9 HugeInt::HugeInt( long value )
10 {
11     // inicializa o array como zero
12     for ( int i = 0; i <= 29; i++ )
13         integer[ i ] = 0;
14
15     // coloca dígitos de argumento no array
16     for ( int j = 29; value != 0 && j >= 0; j-- )
17     {
18         integer[ j ] = value % 10;
19         value /= 10;
20     } // fim do for
21 } // fim do construtor-padrão/de conversão HugeInt
22
23 // construtor de conversão que converte uma string de caractere
24 // representando um inteiro grande em um objeto HugeInt
25 HugeInt::HugeInt( const char *string )
26 {
27     // inicializa o array como zero
28     for ( int i = 0; i <= 29; i++ )
29         integer[ i ] = 0;
30
31     // coloca os dígitos de argumento no array
32     int length = strlen( string );
33
34     for ( int j = 30 - length, k = 0; j <= 29; j++, k++ )
35
36         if ( isdigit( string[ k ] ) )
37             integer[ j ] = string[ k ] - '0';
38 } // fim do construtor de conversão HugeInt

```

Figura 11.23 Definições de função-membro e função friend da classe HugeInt.

(continua)

```

39
40 // operador de adição; HugeInt + HugeInt
41 HugeInt HugeInt::operator+( const HugeInt &op2 ) const
42 {
43     HugeInt temp; // resultado temporário
44     int carry = 0;
45
46     for ( int i = 29; i >= 0; i-- )
47     {
48         temp.integer[ i ] =
49             integer[ i ] + op2.integer[ i ] + carry;
50
51         // determina se transporta ou não um 1
52         if ( temp.integer[ i ] > 9 )
53         {
54             temp.integer[ i ] %= 10; // reduz a 0-9
55             carry = 1;
56         } // fim do if
57         else // não transporta
58             carry = 0;
59     } // fim do for
60
61     return temp; // retorno da cópia do objeto temporário
62 } // fim da função operator+
63
64 // operador de adição; HugeInt + int
65 HugeInt HugeInt::operator+( int op2 ) const
66 {
67     // converte op2 em um HugeInt, então invoca
68     // operator+ para dois objetos HugeInt
69     return *this + HugeInt( op2 );
70 } // fim da função operator+
71
72 // operador de adição;
73 // HugeInt + string que representa o valor de tipo inteiro grande
74 HugeInt HugeInt::operator+( const char *op2 ) const
75 {
76     // converte op2 em um HugeInt, então invoca
77     // operator+ para dois objetos HugeInt
78     return *this + HugeInt( op2 );
79 } // fim de operator+
80
81 // operador de saída sobrecarregado
82 ostream& operator<<( ostream &output, const HugeInt &num )
83 {
84     int i;
85
86     for ( i = 0; ( num.integer[ i ] == 0 ) && ( i <= 29 ); i++ )
87         ; // pula zeros à esquerda
88
89     if ( i == 30 )
90         output << 0;
91     else
92
93         for ( ; i <= 29; i++ )
94             output << num.integer[ i ];

```

Figura 11.23 Definições de função-membro e função friend da classe HugeInt.

(continua)

- 11.15** Crie uma classe `RationalNumber` (frações) com as seguintes capacidades:
- a) Crie um construtor que impede um denominador 0 em uma fração, reduz ou simplifica as frações que não estão na forma reduzida e que evita denominadores negativos.
 - b) Sobrecarregue os operadores de adição, subtração, multiplicação e divisão dessa classe.
 - c) Sobrecarregue os operadores relacionais e de igualdade dessa classe.
- 11.16** Estude as funções da biblioteca de tratamento de strings do C e implemente cada uma das funções como parte da classe `String` (figuras 11.9–11.10). Então, utilize essas funções para realizar manipulações de texto.
- 11.17** Desenvolva a classe `Polynomial`. A representação interna de um `Polynomial` é um array de termos. Cada termo contém um coeficiente e um expoente. O termo
- $$2x^4$$
- tem o coeficiente 2 e o expoente 4. Desenvolva uma classe completa contendo funções construtoras e destrutoras adequadas, bem como as funções *get* e *set*. A classe também deve fornecer as seguintes capacidades de operador sobrecarregado:
- a) Sobrecarregar o operador de adição (+) para adicionar dois `Polynomials`.
 - b) Sobrecarregar o operador de subtração (-) para subtrair dois `Polynomials`.
 - c) Sobrecarregar o operador de atribuição para atribuir um `Polynomial` a outro.
 - d) Sobrecarregar o operador de multiplicação (*) para multiplicar dois `Polynomials`.
 - e) Sobrecarregar o operador de atribuição de adição (+=), o operador de atribuição de subtração (--=) e o operador de atribuição de multiplicação (*=).
- 11.18** No programa das figuras 11.3–11.5, a Figura 11.4 contém o comentário ‘overloaded stream insertion operator; cannot be a member function if we would like to invoke it with `cout << somePhoneNumber;`’ [‘operador de inserção de fluxo sobrecarregado; não pode ser uma função-membro se quisermos invocá-lo com `cout << somePhoneNumber;`’]. De fato, o operador de inserção de fluxo poderia ser uma função-membro da classe `PhoneNumber` se estivéssemos dispostos a invocá-lo como `somePhoneNumber.operator<< ( cout );` ou como `somePhoneNumber << cout;`. Reescreva o programa da Figura 11.5 com a inserção de fluxo `operator<<` sobrecarregado como uma função-membro e experimente as duas instruções precedentes no programa para demonstrar que elas funcionam.



Nunca diga que você conhece inteiramente uma pessoa, até dividir uma herança com ela.
Johann Kasper Lavater

Esse método é para definir como o número de uma classe, a classe de todas as classes similares à classe fornecida.
Bertrand Russell

Melhor que herdar uma biblioteca, é colecionar uma.
Augustine Birrell

Sempre foi desprezível o lucro que nos vem de pergaminho.
William Shakespeare

Programação orientada a objetos: herança

OBJETIVOS

Neste capítulo, você aprenderá:

- A criar classes herdando de classes existentes.
- Como a herança promove a reutilização de software.
- As noções de classes básicas e classes derivadas e os relacionamentos entre elas.
- O especificador de acesso de membro `protected`.
- O uso de construtores e destrutores em hierarquias de herança.
- As diferenças entre herança `public`, `protected` e `private`.
- O uso de herança para personalizar software existente.

- 12.1** Introdução
- 12.2** Classes básicas e derivadas
- 12.3** Membros `protected`
- 12.4** Relacionamento entre classes básicas e derivadas
 - 12.4.1** Criando e utilizando uma classe `CommissionEmployee`
 - 12.4.2** Criando uma classe `BasePlusCommissionEmployee` sem utilizar herança
 - 12.4.3** Criando uma hierarquia de herança `CommissionEmployee–BasePlusCommissionEmployee`
 - 12.4.4** Hierarquia de herança `CommissionEmployee–BasePlusCommissionEmployee` utilizando dados `protected`
 - 12.4.5** Hierarquia de herança `CommissionEmployee–BasePlusCommissionEmployee` utilizando dados `private`
- 12.5** Construtores e destrutores em classes derivadas
- 12.6** Herança `public`, `protected` e `private`
- 12.7** Engenharia de software com herança
- 12.8** Síntese

Resumo | Terminologia | Exercícios de revisão | Respostas dos exercícios de revisão | Exercícios

12.1 Introdução

Este capítulo continua nossa discussão de programação orientada a objetos (*object-oriented programming* – OOP) introduzindo outro de seus recursos-chave — a **herança**. A herança é uma forma de reutilização de software em que o programador cria uma classe que absorve dados e comportamentos de uma classe existente e os aprimora com novas capacidades. A reusabilidade de software economiza tempo durante o desenvolvimento de programa. Ela também encoraja a reutilização de softwares de alta qualidade já testados e depurados, o que aumenta a probabilidade de um sistema ser eficientemente implementado.

Ao criar uma classe, em vez de escrever membros de dados e funções-membro completamente novos, o programador pode designar que a nova classe deve **herdar** membros de uma classe existente. Essa classe existente é chamada **classe básica** e a nova classe é referida como **classe derivada**. (Outras linguagens de programação, como Java, referem-se à classe básica como **superclasse** e à classe derivada como **subclasse**.) Uma classe derivada representa um grupo mais especializado de objetos. Em geral, uma classe derivada contém comportamentos herdados de sua classe básica mais comportamentos adicionais. Como veremos, uma classe derivada também pode personalizar comportamentos herdados da classe básica. Uma **classe básica direta** é a classe básica a partir da qual uma classe derivada herda explicitamente. Uma **classe básica indireta** é herdada de dois ou mais níveis na **hierarquia de classes**. No caso de **herança simples**, uma classe é derivada de uma classe básica. O C++ também suporta **herança múltipla**, em que uma classe derivada herda de múltiplas (possivelmente não relacionadas) classes básicas. A herança simples é simples e direta — mostramos vários exemplos que devem permitir ao leitor tornar-se proficiente rapidamente. A herança múltipla pode ser complexa e propensa a erros. Discutimos a herança múltipla no Capítulo 24, “Outros tópicos”.

O C++ oferece três tipos de herança — `public`, `protected` e `private`. Neste capítulo, concentramos-nos na herança `public` e explicamos brevemente as outras duas. No Capítulo 21, “Estruturas de dados”, mostramos como a herança `private` pode ser utilizada como uma alternativa à composição. A terceira forma, a herança `protected`, é raramente utilizada. Com a herança `public`, todo objeto de uma classe derivada também é um objeto de uma classe básica dessa classe derivada. Mas os objetos de classe básica não são objetos de suas classes derivadas. Por exemplo, se tivermos veículo como uma classe básica e um carro como uma classe derivada, então todos os carros são veículos, mas nem todos os veículos são carros. À medida que continuamos nosso estudo de programação orientada a objetos nos capítulos 12 e 13, tiramos proveito desse relacionamento para realizar algumas manipulações interessantes.

A experiência na criação de sistemas de software indica que quantidades significativas de código lidam com casos especiais intimamente relacionados. Quando os programadores estão preocupados com casos especiais, os detalhes podem obscurecer a visão geral. Com a programação orientada a objetos, os programadores se concentram nos aspectos comuns entre objetos no sistema em vez de nos casos especiais.

Distinguímos entre o **relacionamento é um** e o relacionamento *tem um*. O relacionamento *é um* representa a herança. Em um relacionamento *é um*, um objeto de uma classe derivada também pode ser tratado como um objeto de sua classe básica — por exemplo, um carro *é um* veículo, então quaisquer propriedades e comportamentos de um veículo também são propriedades de um carro. Por contraste, o relacionamento *tem um* representa a composição. (A composição foi discutida no Capítulo 10.) Em um relacionamento *tem um*, um objeto contém um ou mais objetos de outras classes como membros. Por exemplo, um carro inclui muitos componentes — ele *tem uma* direção, *tem um* pedal de freio, *tem um* mecanismo de transmissão e *tem* muitos outros componentes.

As funções-membro da classe derivada talvez requeiram acesso a membros de dados das classes básicas e funções-membro. Uma classe derivada pode acessar os membros não-`private` de sua classe básica. Os membros da classe básica que não devem ser acessíveis às funções-membro das classes derivadas devem ser declarados `private` na classe básica. Uma classe derivada pode produzir alterações

de estado em membros da classe básica `private`, mas somente por funções-membro não-`private` fornecidas na classe básica e herdadas na classe derivada.



Observação de engenharia de software 12.1

As funções-membro de uma classe derivada não podem acessar diretamente os membros `private` da classe básica.



Observação de engenharia de software 12.2

Se uma classe derivada pudesse acessar os membros `private` de sua classe básica, as classes que herdaram dessa classe derivada também poderiam acessar esses dados. Isso propagaria acesso ao que devem ser variáveis de instância `private` e os benefícios do ocultamento de informações seriam perdidos.

Um problema com a herança é que uma classe derivada pode herdar membros de dados e funções-membro de que ela não precisa ou que ela não deve ter. É responsabilidade do projetista da classe assegurar que as capacidades fornecidas por uma classe sejam apropriadas às futuras classes derivadas. Mesmo quando uma função-membro de classe básica for apropriada a uma classe derivada, a classe derivada frequentemente requer que a função-membro se comporte de maneira específica à classe derivada. Nesses casos, a função-membro da classe básica pode ser redefinida na classe derivada com uma implementação apropriada.

12.2 Classes básicas e derivadas

Freqüentemente, um objeto de uma classe também *é um* objeto de outra classe. Por exemplo, em geometria, um retângulo *é um* quadrilátero (assim como o são os quadrados, paralelogramos e trapezóides). Portanto, em C++, pode-se dizer que a classe `Retângulo` *herda* da classe `Quadrilátero`. Nesse contexto, a classe `Quadrilátero` *é uma* classe básica, e a classe `Retângulo` *é uma* classe derivada. Um retângulo *é um* tipo específico de quadrilátero, mas é incorreto afirmar que um quadrilátero *é um* retângulo — o quadrilátero poderia ser um paralelogramo ou alguma outra forma. A Figura 12.1 lista vários exemplos simples de classes básicas e classes derivadas.

Como cada objeto de classe derivada *é um* objeto da sua classe básica e uma classe básica pode ter muitas classes derivadas, o conjunto de objetos representado por uma classe básica é, em geral, maior que o conjunto de objetos representados por qualquer de suas classes derivadas. Por exemplo, a classe básica `Veículo` representa todos os veículos, incluindo carros, caminhões, barcos, aviões, bicicletas e assim por diante. Em comparação, a classe derivada `Carro` representa um subconjunto menor mais específico de todos os veículos.

Os relacionamentos de herança formam estruturas hierárquicas do tipo árvore. Uma classe básica existe em um relacionamento hierárquico com suas classes derivadas. Embora as classes possam existir independentemente, depois que são empregadas em relacionamentos de herança, elas se tornam afiliadas de outras classes. Uma classe torna-se uma classe básica — fornecendo membros para outras classes, ou uma classe derivada — herdando seus membros de outras classes, ou ambas.

Vamos desenvolver uma hierarquia de herança simples com cinco níveis (representados pelo diagrama de classe UML na Figura 12.2). A comunidade de uma universidade tem milhares de membros. Esses membros consistem em empregados, alunos e bacharéis. Os empregados incluem os membros do corpo docente e os funcionários operacionais. Os membros do corpo docente são administradores (como diretores e chefes de departamento) ou professores. Alguns administradores, porém, também são professores. Observe que utilizamos a herança múltipla para formar a classe `ProfessorAdministrador`. Observe também que essa hierarquia de herança poderia conter muitas outras classes. Por exemplo, alunos podem ser graduados ou graduandos. Os alunos graduandos podem ser primeiranistas, segundanistas, terceiranistas e quartanistas.

Cada seta na hierarquia (Figura 12.2) representa um relacionamento *é um*. Por exemplo, enquanto seguimos as setas nessa hierarquia de classes, podemos declarar que ‘um `Empregado` *é um* `MembroDaComunidade`’ e ‘um `Professor` *é um* membro do `CorpoDocente`’. `MembroDaComunidade` é a classe básica direta de `Empregado`, `Aluno` e `Graduados`. Além disso, `MembroDaComunidade` é uma classe básica indireta de todas as outras classes no diagrama. Iniciando da parte inferior do diagrama, o leitor pode seguir as setas e aplicar o relacionamento *é um* à classe básica mais alta. Por exemplo, um `ProfessorAdministrador` *é um* `Administrador`, *é um* membro do `CorpoDocente`, *é um* `Empregado` e *é um* `MembroDaComunidade`.

Classe básica	Classes derivadas
Aluno	AlunoDeGraduação, AlunoDePósGraduação
Forma	Círculo, Triângulo, Retângulo, Esfera, Cubo
Financiamento	FinanciamentoDeCarro, FinanciamentoDeReformaDeCasa, FinanciamentoDeCasaPrópria
Empregado	CorpoDocente, Funcionários
Conta	ContaCorrente, ContaPoupança

Figura 12.1 Exemplos de herança.

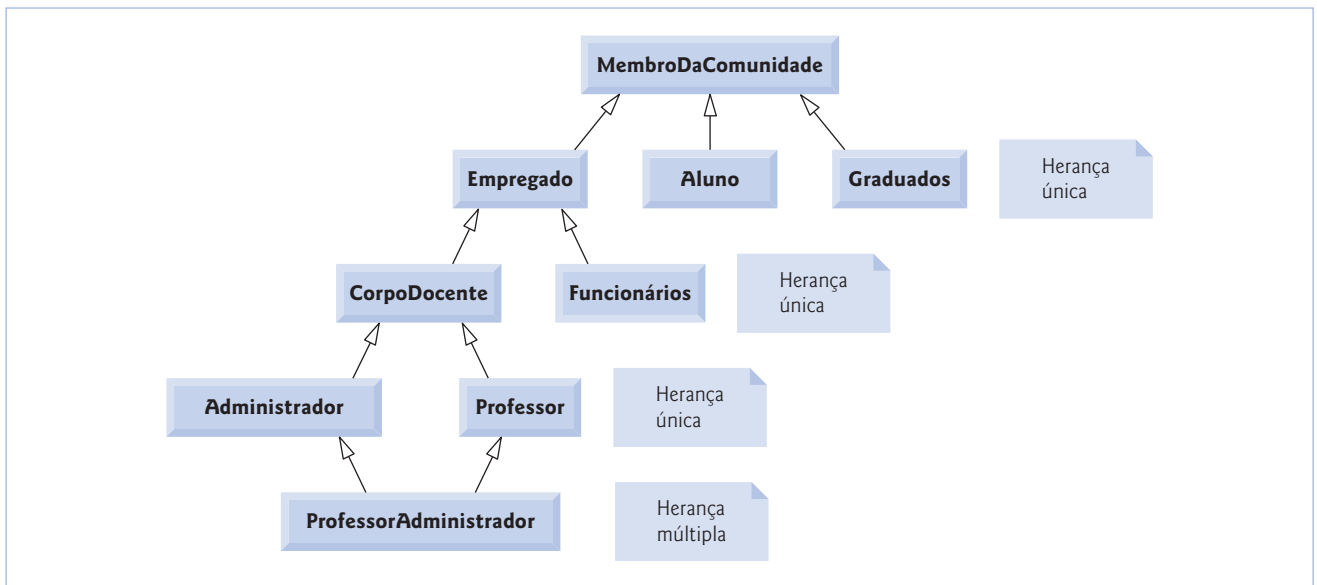


Figura 12.2 Hierarquia de herança de MembrosDaComunidade da universidade.

Agora considere a hierarquia de herança Forma na Figura 12.3. Essa hierarquia começa com a classe básica Forma. As classes FormaBiDimensional e FormaTriDimensional derivam da classe básica Forma — uma Forma é FormaBiDimensional ou FormaTriDimensional. O terceiro nível dessa hierarquia contém alguns tipos mais específicos de FormaBiDimensional e FormaTriDimensional. Como na Figura 12.2, você pode seguir as setas da parte inferior do diagrama para a classe básica superior nessa hierarquia de classes para identificar vários relacionamentos *é um*. Por exemplo, um Triângulo *é uma* FormaBiDimensional e *é uma* Forma, enquanto uma Esfera *é uma* FormaTriDimensional e *é uma* Forma. Observe que essa hierarquia poderia conter muitas outras classes, como Retângulos, Elipses e Trapezóides, os quais são todos FormaBiDimensional.

Especificar que a classe FormaBiDimensional (Figura 12.3) é derivada da (ou herda da) classe Forma, a classe FormaBiDimensional poderia ser definida em C++ como segue:

```
class FormaBiDimensional : public Forma
```

Esse é um exemplo de **herança public**, a forma mais comumente utilizada. Também discutiremos **herança private** e **herança protected** (Seção 12.6). Com todas as formas de herança, os membros **private** de uma classe básica não são acessíveis diretamente de classes derivadas dessa classe, mas esses membros de classe básica **private** ainda são herdados (isto é, ainda são considerados partes das classes derivadas). Com a herança **public**, todos os outros membros de classe básica retêm seu acesso original de membro quando se tornam membros da classe derivada (por exemplo, os membros **public** da classe básica tornam-se membros **public** da classe derivada, e, como veremos em breve, os membros **protected** da classe básica tornam-se os membros **protected** da classe derivada). Por meio desses membros herdados da classe básica, a classe derivada pode manipular membros **private** da classe básica (se esses membros herdados fornecerem tal funcionalidade na classe básica). Observe que as funções **friend** não são herdas.

A herança não é apropriada a todos os relacionamentos de classe. No Capítulo 10, discutimos o relacionamento *tem um*, em que as classes têm membros que são objetos de outras classes. Esses relacionamentos criam classes compondo classes existentes. Por exemplo, dadas as classes Employee, BirthDate e TelephoneNumber, é incorreto dizer que um Employee *é um* BirthDate ou que um Employee

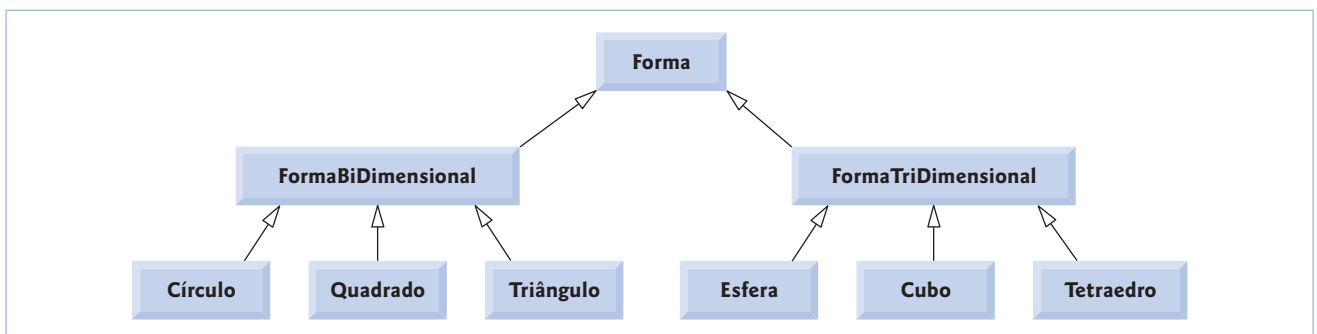


Figura 12.3 Hierarquia de herança para Formas.

é um `PhoneNumber`. Entretanto, é apropriado dizer que um `Employee` *tem uma* `BirthDate` e que um `Employee` *tem um* `PhoneNumber`.

É possível tratar os objetos da classe básica e os objetos da classe derivada de modo semelhante; seus aspectos comuns são expressos nos membros da classe básica. Os objetos de todas as classes derivadas de uma classe básica comum podem ser tratados como objetos dessa classe básica (isto é, esses objetos têm um relacionamento *é um* com a classe básica). No Capítulo 13, “Programação orientada a objetos: polimorfismo”, consideramos muitos exemplos que tiram proveito desse relacionamento.

12.3 Membros `protected`

O Capítulo 3 introduziu os especificadores de acesso `public` e `private`. Os membros `public` de uma classe básica são acessíveis a partir de dentro do corpo dessa classe básica e de qualquer lugar em que o programa tenha um *handle* (isto é, um nome, referência ou ponteiro) para um objeto dessa classe básica ou uma de suas classes derivadas. Os membros `private` de uma classe básica são acessíveis somente dentro do corpo dessa classe básica e dos `friends` dessa classe básica. Nesta seção, introduzimos um especificador de acesso adicional: `protected`.

Utilizar acesso `protected` oferece um nível intermediário de proteção entre `public` e `private`. Os membros `protected` de uma classe básica podem ser acessados a partir de dentro do corpo dessa classe básica, por membros e `friends` dessa classe básica e por membros e `friends` de qualquer classe derivada dessa classe básica.

As funções-membro de classe derivada podem referenciar os membros `public` e `protected` da classe básica simplesmente utilizando os nomes de membro. Quando uma função-membro de classe derivada redefine uma função-membro de uma classe básica, o membro da classe básica pode ser acessado a partir da classe derivada precedendo o nome do membro da classe básica com o nome da classe básica e o operador de resolução de escopo binário (`::`). Discutimos o acesso a membros redefinidos da classe básica na Seção 12.4 e o uso de dados `protected` na Seção 12.4.4.

12.4 Relacionamento entre classes básicas e derivadas

Nesta seção, utilizamos uma hierarquia de herança contendo tipos de empregados no aplicativo de folha de pagamento de uma empresa para discutir o relacionamento entre uma classe básica e uma derivada. Os empregados comissionados (que serão representados como objetos de uma classe básica) são pagos com uma porcentagem de suas vendas, enquanto os empregados comissionados com salário-base (que serão representados como objetos de uma classe derivada) recebem um salário-base mais uma porcentagem de suas vendas. Dividimos nossa discussão do relacionamento entre empregados comissionados e empregados comissionados com salário-base em uma série de cinco exemplos cuidadosamente elaborados passo a passo:

1. No primeiro exemplo, criamos a classe `CommissionEmployee`, que contém como membros de dados `private` nome, sobrenome, número de seguro social, taxa de comissão (porcentagem) e quantidade de vendas brutas (isto é, total).
2. O segundo exemplo define a classe `BasePlusCommissionEmployee`, que contém como membros de dados `private` nome, sobrenome, número de seguro social, taxa de comissão, quantidade de vendas brutas e salário-base. Criamos a última escrevendo cada linha de código que a classe exige — logo veremos que é muito mais eficiente criar essa classe simplesmente herdando da classe `CommissionEmployee`.
3. O terceiro exemplo define uma nova versão da classe `BasePlusCommissionEmployee` que herda diretamente da classe `CommissionEmployee` (isto é, um `BasePlusCommissionEmployee` *é um* `CommissionEmployee` que também tem um salário-base) e tenta acessar membros `private` da classe `CommissionEmployee` — isso resulta em erros de compilação, porque a classe derivada não tem acesso aos dados `private` da classe básica.
4. O quarto exemplo mostra que se os dados do `CommissionEmployee` forem declarados como `protected`, uma nova versão da classe `BasePlusCommissionEmployee` que herda da classe `CommissionEmployee` *pode* acessar esses dados diretamente. Para esse propósito, definimos uma nova versão da classe `CommissionEmployee` com dados `protected`. Ambas as classes `BasePlusCommissionEmployee`, a herdada e a não herdada, contêm funcionalidades idênticas, mas mostramos como a versão de `BasePlusCommissionEmployee` que herda da classe `CommissionEmployee` é mais fácil de criar e gerenciar.
5. Depois de discutirmos a conveniência de utilizar os dados `protected`, criamos o quinto exemplo, que configura os membros de dados `CommissionEmployee` de volta como `private` para impor boa engenharia de software. Esse exemplo demonstra que a classe derivada `BasePlusCommissionEmployee` pode utilizar as funções-membro `public` da classe básica `CommissionEmployee` para manipular os dados `private` de `CommissionEmployee`.

12.4.1 Criando e utilizando uma classe `CommissionEmployee`

Vamos primeiro examinar a definição da classe `CommissionEmployee` (figuras 12.4–12.5). O arquivo de cabeçalho `CommissionEmployee` (Figura 12.4) especifica os serviços `public` da classe `CommissionEmployee` que incluem um construtor (linhas 12–13) e as funções-membro `earnings` (linha 30) e `print` (linha 31). As linhas 15–28 declaram as funções `get` e `set` `public` para manipular os membros de dados da classe (declarados nas linhas 33–37) `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` e `commissionRate`. O arquivo de cabeçalho `CommissionEmployee` especifica cada um desses membros de dados como `private`, então os objetos de outras classes


```

1 // Figura 12.4: CommissionEmployee.h
2 // Classe CommissionEmployee representa um empregado comissionado.
3 #ifndef COMMISSION_H
4 #define COMMISSION_H
5
6 #include <string> // classe string padrão C++
7 using std::string;
8
9 class CommissionEmployee
10 {
11 public:
12     CommissionEmployee( const string &, const string &, const string &,
13                        double = 0.0, double = 0.0 );
14
15     void setFirstName( const string & ); // configura o nome
16     string getFirstName() const; // retorna o nome
17
18     void setLastName( const string & ); // configura o sobrenome
19     string getLastName() const; // retorna o sobrenome
20
21     void setSocialSecurityNumber( const string & ); // configura o SSN
22     string getSocialSecurityNumber() const; // retorna o SSN
23
24     void setGrossSales( double ); // configura a quantidade de vendas brutas
25     double getGrossSales() const; // retorna a quantidade de vendas brutas
26
27     void setCommissionRate( double ); // configura a taxa de comissão (porcentagem)
28     double getCommissionRate() const; // retorna a taxa de comissão
29
30     double earnings() const; // calcula os rendimentos
31     void print() const; // imprime o objeto CommissionEmployee
32 private:
33     string firstName;
34     string lastName;
35     string socialSecurityNumber;
36     double grossSales; // vendas brutas semanais
37     double commissionRate; // porcentagem da comissão
38 }; // fim da classe CommissionEmployee
39
40 #endif

```

Figura 12.4 Arquivo de cabeçalho da classe `CommissionEmployee`.

não podem acessar diretamente esses dados. Declarar membros de dados como `private` e fornecer funções *get* e *set* não-`private` para manipular e validar os membros de dados ajuda a impor boa engenharia de software. As funções-membro `setGrossSales` (definida nas linhas 57–60 da Figura 12.5) e `setCommissionRate` (definida nas linhas 69–72 da Figura 12.5), por exemplo, validam seus argumentos antes de atribuir os valores aos membros de dados `grossSales` e `commissionRate`, respectivamente.

A definição do construtor `CommissionEmployee` não utiliza propositadamente a sintaxe de inicializador de membro nos primeiros vários exemplos desta seção, de modo que podemos demonstrar como os especificadores `private` e `protected` afetam o acesso de membro em classe derivada. Como mostrado na Figura 12.5, linhas 13–15, atribuímos valores aos membros de dados `firstName`, `lastName` e `socialSecurityNumber` no corpo do construtor. Mais tarde nesta seção, retornaremos ao uso de listas de inicializadores de membro nos construtores.

Observe que não validamos os valores dos argumentos do construtor `first`, `last` e `ssn` antes de atribuí-los aos membros de dados correspondentes. Certamente poderíamos validar o nome e o sobrenome — talvez assegurando que eles tenham um comprimento razoável. De maneira semelhante, um SSN poderia ser validado para assegurar que ele contém nove dígitos, com ou sem traços (por exemplo, 123-45-6789 ou 123456789).

A função-membro `earnings` (linhas 81–84) calcula os rendimentos de um `CommissionEmployee`. A linha 83 multiplica `commissionRate` por `grossSales` e retorna o resultado. A função-membro `print` (linhas 87–93) exibe os valores dos membros de dados de um objeto `CommissionEmployee`.

A Figura 12.6 testa a classe `CommissionEmployee`. As linhas 16–17 instanciam o objeto `employee` da classe `CommissionEmployee` e invocam o construtor `CommissionEmployee` para inicializar o objeto com "Sue" como o nome, "Jones" como o sobrenome, "222-22-2222" como o número de seguro social, 10000 como a quantidade de vendas brutas e .06 como a taxa de comissão. As linhas 23–29 utilizam as funções `get` de `employee` para exibir os valores de seus membros de dados. As linhas 31–32 invocam as funções-membro `setGrossSales` e `setCommissionRate` do objeto para alterar os valores dos membros de dados `grossSales` e `commissionRate`, respectivamente. A linha 36 então chama a função-membro `print` de `employee` para gerar saída das informações sobre `CommissionEmployee` atualizadas. Por fim, a linha 39 exibe os rendimentos de `CommissionEmployee`, calculados pela função-membro `earnings` do objeto utilizando os valores atualizados dos membros de dados `grossSales` e `commissionRate`.

```

1 // Figura 12.5: CommissionEmployee.cpp
2 // Definições de função-membro da classe CommissionEmployee.
3 #include <iostream>
4 using std::cout;
5
6 #include "CommissionEmployee.h" // Definição da classe CommissionEmployee
7
8 // construtor
9 CommissionEmployee::CommissionEmployee(
10     const string &first, const string &last, const string &ssn,
11     double sales, double rate )
12 {
13     firstName = first; // deve validar
14     lastName = last;   // deve validar
15     socialSecurityNumber = ssn; // deve validar
16     setGrossSales( sales ); // valida e armazena as vendas brutas
17     setCommissionRate( rate ); // valida e armazena a taxa de comissão
18 } // fim do construtor CommissionEmployee
19
20 // configura o nome
21 void CommissionEmployee::setFirstName( const string &first )
22 {
23     firstName = first; // deve validar
24 } // fim da função setFirstName
25
26 // retorna o nome
27 string CommissionEmployee::getFirstName() const
28 {
29     return firstName;
30 } // fim da função getFirstName
31
32 // configura o sobrenome
33 void CommissionEmployee::setLastName( const string &last )
34 {
35     lastName = last; // deve validar
36 } // fim da função setLastName
37
38 // retorna o sobrenome
39 string CommissionEmployee::getLastName() const
40 {
41     return lastName;
42 } // fim da função getLastName

```

Figura 12.5 O arquivo de implementação para a classe `CommissionEmployee` que representa um empregado que ganha uma porcentagem das vendas brutas. (continua)

```

43
44 // configura o SSN
45 void CommissionEmployee::setSocialSecurityNumber( const string &ssn )
46 {
47     socialSecurityNumber = ssn; // deve validar
48 } // fim da função setSocialSecurityNumber
49
50 // retorna o SSN
51 string CommissionEmployee::getSocialSecurityNumber() const
52 {
53     return socialSecurityNumber;
54 } // fim da função getSocialSecurityNumber
55
56 // configura a quantidade de vendas brutas
57 void CommissionEmployee::setGrossSales( double sales )
58 {
59     grossSales = ( sales < 0.0 ) ? 0.0 : sales;
60 } // fim da função setGrossSales
61
62 // retorna a quantidade de vendas brutas
63 double CommissionEmployee::getGrossSales() const
64 {
65     return grossSales;
66 } // fim da função getGrossSales
67
68 // configura a taxa de comissão
69 void CommissionEmployee::setCommissionRate( double rate )
70 {
71     commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
72 } // fim da função setCommissionRate
73
74 // retorna a taxa de comissão
75 double CommissionEmployee::getCommissionRate() const
76 {
77     return commissionRate;
78 } // fim da função getCommissionRate
79
80 // calcula os rendimentos
81 double CommissionEmployee::earnings() const
82 {
83     return commissionRate * grossSales;
84 } // fim da função earnings
85
86 // imprime o objeto CommissionEmployee
87 void CommissionEmployee::print() const
88 {
89     cout << "commission employee: " << firstName << ' ' << lastName
90         << "\nsocial security number: " << socialSecurityNumber
91         << "\ngross sales: " << grossSales
92         << "\ncommission rate: " << commissionRate;
93 } // fim da função print

```

Figura 12.5 O arquivo de implementação para a classe `CommissionEmployee` que representa um empregado que ganha uma porcentagem das vendas brutas. (continuação)

```

1 // Figura 12.6: fig12_06.cpp
2 // Testando a classe CommissionEmployee.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::fixed;
7
8 #include <iomanip>
9 using std::setprecision;
10
11 #include "CommissionEmployee.h" // Definição da classe CommissionEmployee
12
13 int main()
14 {
15     // instancia um objeto CommissionEmployee
16     CommissionEmployee employee(
17         "Sue", "Jones", "222-22-2222", 10000, .06 );
18
19     // configura a formatação de saída de ponto flutuante
20     cout << fixed << setprecision( 2 );
21
22     // obtém os dados do empregado comissionado
23     cout << "Employee information obtained by get functions: \n"
24         << "\nFirst name is " << employee.getFirstName()
25         << "\nLast name is " << employee.getLastName()
26         << "\nSocial security number is "
27         << employee.getSocialSecurityNumber()
28         << "\nGross sales is " << employee.getGrossSales()
29         << "\nCommission rate is " << employee.getCommissionRate() << endl;
30
31     employee.setGrossSales( 8000 ); // configura as vendas brutas
32     employee.setCommissionRate( .1 ); // configura a taxa de comissão
33
34     cout << "\nUpdated employee information output by print function: \n"
35         << endl;
36     employee.print(); // exibe as novas informações do empregado
37
38     // exibe os rendimentos do empregado
39     cout << "\n\nEmployee's earnings: $" << employee.earnings() << endl;
40
41     return 0;
42 } // fim de main

```

Employee information obtained by get functions:

First name is Sue
 Last name is Jones
 Social security number is 222-22-2222
 Gross sales is 10000.00
 Commission rate is 0.06

Updated employee information output by print function:

commission employee: Sue Jones
 social security number: 222-22-2222
 gross sales: 8000.00

Figura 12.6 Programa de teste da classe CommissionEmployee.

(continua)

```
commission rate: 0.10

Employee's earnings: $800.00
```

Figura 12.6 Programa de teste da classe `CommissionEmployee`.

(continuação)

12.4.2 Criando uma classe `BasePlusCommissionEmployee` sem utilizar herança

Agora discutimos a segunda parte da nossa introdução à herança criando e testando uma classe (completamente nova e independente) `BasePlusCommissionEmployee` (figuras 12.7–12.8), que contém nome, sobrenome, número de seguro social, quantidade de vendas brutas, taxa de comissão e salário-base.

Definindo a classe `BasePlusCommissionEmployee`

O arquivo de cabeçalho `BasePlusCommissionEmployee` (Figura 12.7) especifica os serviços public da classe `BasePlusCommissionEmployee` que incluem o construtor `BasePlusCommissionEmployee` (linhas 13–14) e as funções-membro `earnings` (linha 34) e `print` (linha 35). As linhas 16–32 declaram as funções `get` e `set` public para membros de dados private (declarados nas linhas 37–42) `firstName`, `lastName`, `socialSecurityNumber`, `grossSales`, `commissionRate` e `baseSalary` da classe. Essas variáveis e funções-membro encapsulam todos os recursos necessários de um empregado comissionado com salário-base. Observe a semelhança entre essa classe e a classe `CommissionEmployee` (figuras 12.4–12.5) — neste exemplo, ainda não exploraremos essa semelhança.

A função-membro `earnings` (definida nas linhas 96–99 da Figura 12.8) da classe `BasePlusCommissionEmployee` computa os rendimentos de um empregado comissionado com salário-base. A linha 98 retorna o resultado da adição do salário-base do empregado ao produto da taxa de comissão e das vendas brutas do empregado.

```
1  // Figura 12.7: BasePlusCommissionEmployee.h
2  // Definição da classe BasePlusCommissionEmployee representa um empregado
3  // que recebe um salário-base além da comissão.
4  #ifndef BASEPLUS_H
5  #define BASEPLUS_H
6
7  #include <string> // classe string padrão C++
8  using std::string;
9
10 class BasePlusCommissionEmployee
11 {
12 public:
13     BasePlusCommissionEmployee( const string &, const string &,
14                                const string &, double = 0.0, double = 0.0, double = 0.0 );
15
16     void setFirstName( const string & ); // configura o nome
17     string getFirstName() const; // retorna o nome
18
19     void setLastName( const string & ); // configura o sobrenome
20     string getLastName() const; // retorna o sobrenome
21
22     void setSocialSecurityNumber( const string & ); // configura o SSN
23     string getSocialSecurityNumber() const; // retorna o SSN
24
25     void setGrossSales( double ); // configura a quantidade de vendas brutas
26     double getGrossSales() const; // retorna a quantidade de vendas brutas
27
28     void setCommissionRate( double ); // configura a taxa de comissão
29     double getCommissionRate() const; // retorna a taxa de comissão
30
```

Figura 12.7 Arquivo de cabeçalho da classe `BasePlusCommissionEmployee`.

(continua)

```

31 void setBaseSalary( double ); // configura o salário-base
32 double getBaseSalary() const; // retorna o salário-base
33
34 double earnings() const; // calcula os rendimentos
35 void print() const; // imprime o objeto BasePlusCommissionEmployee
36 private:
37     string firstName;
38     string lastName;
39     string socialSecurityNumber;
40     double grossSales; // vendas brutas semanais
41     double commissionRate; // porcentagem da comissão
42     double baseSalary; // salário-base
43 }; // fim da classe BasePlusCommissionEmployee
44
45 #endif

```

Figura 12.7 Arquivo de cabeçalho da classe BasePlusCommissionEmployee.

(continuação)

```

1 // Figura 12.8: BasePlusCommissionEmployee.cpp
2 // Definições de função-membro da classe BasePlusCommissionEmployee.
3 #include <iostream>
4 using std::cout;
5
6 // definição da classe BasePlusCommissionEmployee
7 #include "BasePlusCommissionEmployee.h"
8
9 // construtor
10 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
11     const string &first, const string &last, const string &ssn,
12     double sales, double rate, double salary )
13 {
14     firstName = first; // deve validar
15     lastName = last; // deve validar
16     socialSecurityNumber = ssn; // deve validar
17     setGrossSales( sales ); // valida e armazena as vendas brutas
18     setCommissionRate( rate ); // valida e armazena a taxa de comissão
19     setBaseSalary( salary ); // valida e armazena salário-base
20 } // fim do construtor BasePlusCommissionEmployee
21
22 // configura o nome
23 void BasePlusCommissionEmployee::setFirstName( const string &first )
24 {
25     firstName = first; // deve validar
26 } // fim da função setFirstName
27
28 // retorna o nome
29 string BasePlusCommissionEmployee::getFirstName() const
30 {
31     return firstName;
32 } // fim da função getFirstName
33
34 // configura o sobrenome
35 void BasePlusCommissionEmployee::setLastName( const string &last )
36 {

```

Figura 12.8 A classe BasePlusCommissionEmployee representa um empregado que recebe um salário-base além de uma comissão. (continua)

```

37     lastName = last; // deve validar
38 } // fim da função setLastName
39
40 // retorna o sobrenome
41 string BasePlusCommissionEmployee::getLastName() const
42 {
43     return lastName;
44 } // fim da função getLastName
45
46 // configura o SSN
47 void BasePlusCommissionEmployee::setSocialSecurityNumber(
48     const string &ssn )
49 {
50     socialSecurityNumber = ssn; // deve validar
51 } // fim da função setSocialSecurityNumber
52
53 // retorna o SSN
54 string BasePlusCommissionEmployee::getSocialSecurityNumber() const
55 {
56     return socialSecurityNumber;
57 } // fim da função getSocialSecurityNumber
58
59 // configura a quantidade de vendas brutas
60 void BasePlusCommissionEmployee::setGrossSales( double sales )
61 {
62     grossSales = ( sales < 0.0 ) ? 0.0 : sales;
63 } // fim da função setGrossSales
64
65 // retorna a quantidade de vendas brutas
66 double BasePlusCommissionEmployee::getGrossSales() const
67 {
68     return grossSales;
69 } // fim da função getGrossSales
70
71 // configura a taxa de comissão
72 void BasePlusCommissionEmployee::setCommissionRate( double rate )
73 {
74     commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
75 } // fim da função setCommissionRate
76
77 // retorna a taxa de comissão
78 double BasePlusCommissionEmployee::getCommissionRate() const
79 {
80     return commissionRate;
81 } // fim da função getCommissionRate
82
83 // configura o salário-base
84 void BasePlusCommissionEmployee::setBaseSalary( double salary )
85 {
86     baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
87 } // fim da função setBaseSalary
88
89 // retorna o salário-base
90 double BasePlusCommissionEmployee::getBaseSalary() const
91 {
92     return baseSalary;
93 } // fim da função getBaseSalary

```

Figura 12.8 A classe `BasePlusCommissionEmployee` representa um empregado que recebe um salário-base além de uma comissão. (continua)


```

94
95 // calcula os rendimentos
96 double BasePlusCommissionEmployee::earnings() const
97 {
98     return baseSalary + ( commissionRate * grossSales );
99 } // fim da função earnings
100
101 // imprime o objeto BasePlusCommissionEmployee
102 void BasePlusCommissionEmployee::print() const
103 {
104     cout << "base-salaried commission employee: " << firstName << ' '
105         << lastName << "\nsocial security number: " << socialSecurityNumber
106         << "\ngross sales: " << grossSales
107         << "\ncommission rate: " << commissionRate
108         << "\nbase salary: " << baseSalary;
109 } // fim da função print

```

Figura 12.8 A classe `BasePlusCommissionEmployee` representa um empregado que recebe um salário-base além de uma comissão. (continuação)

Testando a classe `BasePlusCommissionEmployee`

A Figura 12.9 testa a classe `BasePlusCommissionEmployee`. As linhas 17–18 instanciam o objeto `employee` da classe `BasePlusCommissionEmployee`, passando "Bob", "Lewis", "333-33-3333", 5000, .04 e 300 para o construtor como nome, sobrenome, número de seguro social, vendas brutas, taxa de comissão e salário-base, respectivamente. As linhas 24–31 utilizam as funções `get` do `BasePlusCommissionEmployee` para recuperar os valores dos membros de dados do objeto para a saída. A linha 33 invoca função-membro `setBaseSalary` do objeto para alterar o salário-base. A função-membro `setBaseSalary` (Figura 12.8, linhas 84–87) assegura que o membro de dados `baseSalary` não receba um valor negativo, porque o salário-base de um empregado não pode ser negativo. A linha 37 da Figura 12.9 invoca a função-membro `print` do objeto para gerar saída de informações `BasePlusCommissionEmployee` atualizadas, e a linha 40 chama a função-membro `earnings` para exibir os rendimentos do `BasePlusCommissionEmployee`.

```

1 // Figura 12.9: fig12_09.cpp
2 // Testando a classe BasePlusCommissionEmployee.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::fixed;
7
8 #include <iomanip>
9 using std::setprecision;
10
11 // definição da classe BasePlusCommissionEmployee
12 #include "BasePlusCommissionEmployee.h"
13
14 int main()
15 {
16     // instancia o objeto BasePlusCommissionEmployee
17     BasePlusCommissionEmployee
18         employee( "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
19
20     // configura a formatação de saída de ponto flutuante
21     cout << fixed << setprecision( 2 );
22
23     // obtém os dados do empregado comissionado

```

Figura 12.9 Programa de teste da classe `BasePlusCommissionEmployee`.

(continua)

```

24     cout << "Employee information obtained by get functions: \n"
25         << "\nFirst name is " << employee.getFirstName()
26         << "\nLast name is " << employee.getLastName()
27         << "\nSocial security number is "
28         << employee.getSocialSecurityNumber()
29         << "\nGross sales is " << employee.getGrossSales()
30         << "\nCommission rate is " << employee.getCommissionRate()
31         << "\nBase salary is " << employee.getBaseSalary() << endl;
32
33     employee.setBaseSalary( 1000 ); // configura o salário-base
34
35     cout << "\nUpdated employee information output by print function: \n"
36         << endl;
37     employee.print(); // exibe as novas informações do empregado
38
39     // exibe os rendimentos do empregado
40     cout << "\n\nEmployee's earnings: $" << employee.earnings() << endl;
41
42     return 0;
43 } // fim de main

```

Employee information obtained by get functions:

First name is Bob
 Last name is Lewis
 Social security number is 333-33-3333
 Gross sales is 5000.00
 Commission rate is 0.04
 Base salary is 300.00

Updated employee information output by print function:

base-salaried commission employee: Bob Lewis
 social security number: 333-33-3333
 gross sales: 5000.00
 commission rate: 0.04
 base salary: 1000.00

Employee's earnings: \$1200.00

Figura 12.9 Programa de teste da classe `BasePlusCommissionEmployee`.

(continuação)

Explorando as semelhanças entre a classe *BasePlusCommissionEmployee* e a classe *CommissionEmployee*

Observe que grande parte do código para a classe `BasePlusCommissionEmployee` (figuras 12.7–12.8) é semelhante, se não idêntica, ao do código para a classe `CommissionEmployee` (figuras 12.4–12.5). Por exemplo, na classe `BasePlusCommissionEmployee`, os membros de dados `private firstName` e `lastName` e as funções-membro `setFirstName`, `getFirstName`, `setLastName` e `getLastName` são idênticos aos da classe `CommissionEmployee`. As classes `CommissionEmployee` e `BasePlusCommissionEmployee` também contêm, ambas, os membros de dados `private socialSecurityNumber`, `commissionRate` e `grossSales`, bem como as funções *get* e *set* para manipular esses membros. Além disso, o construtor `BasePlusCommissionEmployee` é quase idêntico àquele da classe `CommissionEmployee`, exceto pelo fato de que o construtor de `BasePlusCommissionEmployee` também configura o `baseSalary`. As outras adições à classe `BasePlusCommissionEmployee` são os membros de dados `private baseSalary` e as funções-membro `setBaseSalary` e `getBaseSalary`. A função-membro `print` da classe `BasePlusCommissionEmployee` é quase idêntica à da classe `CommissionEmployee`, exceto pelo fato de que `print` de `BasePlusCommissionEmployee` também gera saída do valor do membro de dados `baseSalary`.

Copiamos literalmente o código da classe `CommissionEmployee` e o colamos na classe `BasePlusCommissionEmployee`, então modificamos a classe `BasePlusCommissionEmployee` para incluir um salário-base e as funções-membro que manipulam o salário-base. Essa abordagem ‘copiar e colar’ é freqüentemente propensa a erro e demorada. Pior ainda, ela pode espalhar muitas cópias físicas do mesmo

código por todo um sistema, criando um pesadelo para a manutenção de código. Existe um modo de ‘absorver’ os membros de dados e funções-membro de uma classe de maneira que façam parte de outras classes sem duplicar o código? Nos próximos vários exemplos, fazemos exatamente isso, utilizando a herança.



Observação de engenharia de software 12.3

Copiar e colar código de uma classe para a outra pode espalhar erros por múltiplos arquivos de código-fonte. Para evitar a duplicação de código (e possivelmente erros), utilize a herança, em vez da abordagem ‘copiar e colar’, em situações em que você quer que uma classe ‘absorva’ os membros de dados e as funções-membro de outra classe.



Observação de engenharia de software 12.4

Com a herança, os membros de dados e funções-membro comuns a todas as classes na hierarquia são declarados em uma classe básica. Quando esses recursos comuns requerem mudanças, os desenvolvedores de software precisam fazer as alterações somente na classe básica — as classes derivadas herdam as alterações. Sem a herança, as alterações precisariam ser feitas em todos os arquivos de código-fonte que contêm uma cópia do código em questão.

12.4.3 Criando uma hierarquia de herança `CommissionEmployee–BasePlusCommissionEmployee`

Agora criamos e testamos uma nova versão da classe `BasePlusCommissionEmployee` (figuras 12.10–12.11) que deriva da classe `CommissionEmployee` (figuras 12.4–12.5). Nesse exemplo, um objeto `BasePlusCommissionEmployee` é um `CommissionEmployee` (porque a herança transfere as capacidades da classe `CommissionEmployee`), mas a classe `BasePlusCommissionEmployee` também tem o membro de dados `baseSalary` (Figura 12.10, linha 24). O sinal de dois-pontos (:) na linha 12 da definição de classe indica herança. A palavra-chave `public` indica o tipo de herança. Como uma classe derivada (formada com a herança `public`), `BasePlusCommissionEmployee` herda todos os membros da classe `CommissionEmployee`, exceto pelo construtor — cada classe fornece seus próprios construtores que são específicos à classe. [Observe que os destrutores também não são herdados.] Portanto, os serviços `public` de `BasePlusCommissionEmployee` incluem seu construtor (linhas 15–16) e as funções-membro `public` herdadas da classe `CommissionEmployee` — embora

```

1 // Figura 12.10: BasePlusCommissionEmployee.h
2 // Classe BasePlusCommissionEmployee derivada da classe
3 // CommissionEmployee.
4 #ifndef BASEPLUS_H
5 #define BASEPLUS_H
6
7 #include <string> // classe string padrão C++
8 using std::string;
9
10 #include "CommissionEmployee.h" // declaração da classe CommissionEmployee
11
12 class BasePlusCommissionEmployee : public CommissionEmployee
13 {
14 public:
15     BasePlusCommissionEmployee( const string &, const string &,
16                               const string &, double = 0.0, double = 0.0, double = 0.0 );
17
18     void setBaseSalary( double ); // configura o salário-base
19     double getBaseSalary() const; // retorna o salário-base
20
21     double earnings() const; // calcula os rendimentos
22     void print() const; // imprime o objeto BasePlusCommissionEmployee
23 private:
24     double baseSalary; // salário-base
25 }; // fim da classe BasePlusCommissionEmployee
26
27 #endif

```

Figura 12.10 A definição da classe `BasePlusCommissionEmployee` indicando o relacionamento de herança com a classe `CommissionEmployee`.

não possamos ver essas funções-membro herdadas no código-fonte de `BasePlusCommissionEmployee`, elas são, contudo, uma parte da classe derivada `BasePlusCommissionEmployee`. Os serviços públicos da classe derivada também incluem as funções-membro `setBaseSalary`, `getBaseSalary`, `earnings` e `print` (linhas 18–22).

A Figura 12.11 mostra as implementações da função-membro `BasePlusCommissionEmployee`. O construtor (linhas 10–17) introduz a **sintaxe inicializadora da classe básica** (linha 14), que utiliza um inicializador de membro para argumentos passados ao construtor da classe básica (`CommissionEmployee`). O C++ requer que o construtor de classe derivada chame seu construtor de classe básica para inicializar os membros de dados da classe básica que são herdados na classe derivada. A linha 14 realiza essa tarefa invocando o construtor `CommissionEmployee` pelo nome, passando os parâmetros do construtor `first`, `last`, `ssn`, `sales` e `rate` como argumentos para inicializar os membros de dados da classe básica `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` e `commissionRate`. Se o construtor `BasePlusCommissionEmployee` não invocasse o construtor da classe `CommissionEmployee` explicitamente, o C++ tentaria invocar o construtor-padrão da classe `CommissionEmployee` — mas a classe não tem esse construtor, portanto o compilador emitiria um erro. Considerando a discussão do Capítulo 3, lembre-se de que o compilador fornece um construtor-padrão sem parâmetros em qualquer classe que não inclui explicitamente um construtor. Entretanto, `CommissionEmployee` inclui um construtor explicitamente, então um construtor-padrão não é fornecido e qualquer tentativa de chamar implicitamente o construtor-padrão do `CommissionEmployee` resultaria em erros de compilação.

```

1 // Figura 12.11: BasePlusCommissionEmployee.cpp
2 // Definições de função-membro da classe BasePlusCommissionEmployee.
3 #include <iostream>
4 using std::cout;
5
6 // definição da classe BasePlusCommissionEmployee
7 #include "BasePlusCommissionEmployee.h"
8
9 // construtor
10 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
11     const string &first, const string &last, const string &ssn,
12     double sales, double rate, double salary )
13     // chama explicitamente o construtor da classe básica
14     : CommissionEmployee( first, last, ssn, sales, rate )
15 {
16     setBaseSalary( salary ); // valida e armazena salário-base
17 } // fim do construtor BasePlusCommissionEmployee
18
19 // configura o salário-base
20 void BasePlusCommissionEmployee::setBaseSalary( double salary )
21 {
22     baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
23 } // fim da função setBaseSalary
24
25 // retorna o salário-base
26 double BasePlusCommissionEmployee::getBaseSalary() const
27 {
28     return baseSalary;
29 } // fim da função getBaseSalary
30
31 // calcula os rendimentos
32 double BasePlusCommissionEmployee::earnings() const
33 {
34     // a classe derivada não pode acessar dados private da classe básica
35     return baseSalary + ( commissionRate * grossSales );
36 } // fim da função earnings
37
38 // imprime o objeto BasePlusCommissionEmployee

```

Figura 12.11 Arquivo de implementação de `BasePlusCommissionEmployee`: dados `private` da classe básica não podem ser acessados a partir da classe derivada. (continua)

```

39 void BasePlusCommissionEmployee::print() const
40 {
41     // a classe derivada não pode acessar dados private da classe básica
42     cout << "base-salaried commission employee: " << firstName << ' '
43         << lastName << "\nsocial security number: " << socialSecurityNumber
44         << "\ngross sales: " << grossSales
45         << "\ncommission rate: " << commissionRate
46         << "\nbase salary: " << baseSalary;
47 } // fim da função print

```

```

C:\cpphttp5_examples\ch12\Fig12_10_11\BasePlusCommission-Employee.cpp(35) :
error C2248: 'CommissionEmployee::commissionRate' :
cannot access private member declared in class 'CommissionEmployee'
C:\cpphttp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(37) :
see declaration of 'CommissionEmployee::commissionRate'
C:\cpphttp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(10) :
see declaration of 'CommissionEmployee'

C:\cpphttp5_examples\ch12\Fig12_10_11\BasePlusCommission-Employee.cpp(35) :
error C2248: 'CommissionEmployee::grossSales' :
cannot access private member declared in class 'CommissionEmployee'
C:\cpphttp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(36) :
see declaration of 'CommissionEmployee::grossSales'
C:\cpphttp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(10) :
see declaration of 'CommissionEmployee'

C:\cpphttp5_examples\ch12\Fig12_10_11\BasePlusCommission-Employee.cpp(42) :
error C2248: 'CommissionEmployee::firstName' :
cannot access private member declared in class 'CommissionEmployee'
C:\cpphttp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(33) :
see declaration of 'CommissionEmployee::firstName'
C:\cpphttp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(10) :
see declaration of 'CommissionEmployee'

C:\cpphttp5_examples\ch12\Fig12_10_11\BasePlusCommission-Employee.cpp(43) :
error C2248: 'CommissionEmployee::lastName' :
cannot access private member declared in class 'CommissionEmployee'
C:\cpphttp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(34) :
see declaration of 'CommissionEmployee::lastName'
C:\cpphttp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(10) :
see declaration of 'CommissionEmployee'

C:\cpphttp5_examples\ch12\Fig12_10_11\BasePlusCommission-Employee.cpp(43) :
error C2248: 'CommissionEmployee::socialSecurityNumber' :
cannot access private member declared in class 'CommissionEmployee'
C:\cpphttp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(35) :
see declaration of 'CommissionEmployee::socialSecurityNumber'
C:\cpphttp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(10) :
see declaration of 'CommissionEmployee'

C:\cpphttp5_examples\ch12\Fig12_10_11\BasePlusCommission-Employee.cpp(44) :
error C2248: 'CommissionEmployee::grossSales' :
cannot access private member declared in class 'CommissionEmployee'
C:\cpphttp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(36) :
see declaration of 'CommissionEmployee::grossSales'

```

Figura 12.11

Arquivo de implementação de BasePlusCommissionEmployee: dados private da classe básica não podem ser acessados a partir da classe derivada. (continua)

```

C:\cpphttp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(10) :
    see declaration of 'CommissionEmployee'

C:\cpphttp5_examples\ch12\Fig12_10_11\BasePlusCommissionEmployee.cpp(45) :
    error C2248: 'CommissionEmployee::commissionRate' :
    cannot access private member declared in class 'CommissionEmployee'
    C:\cpphttp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(37) :
        see declaration of 'CommissionEmployee::commissionRate'
    C:\cpphttp5_examples\ch12\Fig12_10_11\CommissionEmployee.h(10) :
        see declaration of 'CommissionEmployee'

```

Figura 12.11 Arquivo de implementação de BasePlusCommissionEmployee: dados private da classe básica não podem ser acessados a partir da classe derivada. (continuação)



Erro comum de programação 12.1

Ocorrerá um erro de compilação se um construtor de classe derivada chamar um de seus construtores de classe básica com argumentos que são inconsistentes com o número e os tipos de parâmetros especificados em uma das definições de construtor de classe básica.



Dica de desempenho 12.1

Em um construtor de classe derivada, inicializar os objetos-membro e invocar construtores de classe básica explicitamente na lista de inicializadores de membro impede a inicialização duplicada na qual um construtor-padrão é chamado, então os membros de dados são modificados novamente no corpo do construtor da classe derivada.

O compilador gera erros para a linha 35 da Figura 12.11 porque os membros de dados `commissionRate` e `grossSales` da classe básica `CommissionEmployee` são `private` — funções-membro da classe derivada `BasePlusCommissionEmployee` não têm permissão de acessar os dados `private` da classe básica `CommissionEmployee`. Observe que utilizamos texto em *itálico* na Figura 12.11 para indicar o código incorreto. O compilador emite erros adicionais nas linhas 42–45 da função-membro `print` da classe `BasePlusCommissionEmployee` pela mesma razão. Como você pode ver, o C++ impõe restrições rígidas no acesso a membros de dados `private`, para que até mesmo uma classe derivada (que é intimamente relacionada com sua classe básica) não possa acessar os dados `private` da classe básica. [Nota: Para poupar espaço, mostramos somente as mensagens de erro do Visual C++ .NET neste exemplo. As mensagens de erro produzidas por seu compilador podem diferir das mostradas aqui. Note também que destacamos em **negrito** as partes-chave das longas mensagens de erro.]

Incluimos propositamente o código errado na Figura 12.11 para demonstrar que funções-membro de uma classe derivada não podem acessar os dados `private` de sua classe básica. Os erros em `BasePlusCommissionEmployee` poderiam ser evitados utilizando as funções-membro `get` herdadas da classe `CommissionEmployee`. Por exemplo, a linha 35 poderia ter invocado `getCommissionRate` e `getGrossSales` para acessar, respectivamente, os membros de dados `private` `commissionRate` e `grossSales` da classe `CommissionEmployee`. De maneira semelhante, as linhas 42–45 poderiam utilizar as funções-membro `get` apropriadas para recuperar os valores dos membros de dados da classe básica. No próximo exemplo, mostramos que usar os dados `protected` também permite evitar os erros encontrados nesse exemplo.

*Incluindo o arquivo de cabeçalho da classe básica no arquivo de cabeçalho da classe derivada com **#include***

Note que incluímos (`#include`) o arquivo de cabeçalho da classe básica no arquivo de cabeçalho da classe derivada (linha 10 da Figura 12.10). Essa é necessária por três razões. Primeiro, para a classe derivada utilizar o nome da classe básica na linha 12, devemos dizer ao compilador que a classe básica existe — a definição de classe em `CommissionEmployee.h` faz exatamente isso.

A segunda razão é que o compilador utiliza uma definição de classe para determinar o tamanho de um objeto dessa classe (como discutimos na Seção 3.8). Um programa-cliente que cria um objeto de uma classe deve incluir (`#include`) a definição de classe para permitir ao compilador reservar a quantidade adequada de memória para o objeto. Ao utilizar herança, o tamanho de um objeto de classe derivada depende dos membros de dados declarados explicitamente em sua definição de classe e dos membros de dados herdados de suas classes básicas diretas e indiretas. Incluir a definição da classe básica na linha 10 permite ao compilador determinar os requisitos de memória para os membros de dados da classe básica que se tornam parte de um objeto de classe derivada e, assim, contribuem para o tamanho total do objeto de classe derivada.

A última razão da linha 10 é permitir ao compilador determinar se a classe derivada utiliza ou não os membros herdados da classe básica adequadamente. Por exemplo, no programa das figuras 12.10–12.11, o compilador utiliza o arquivo de cabeçalho da classe básica para determinar que os membros de dados sendo acessados pela classe derivada são `private` na classe básica. Visto que esses são inacessíveis à classe derivada, o compilador gera erros. O compilador também utiliza os protótipos de função da classe básica para validar

as chamadas de função feitas pela classe derivada para as funções de classe básica herdadas — você verá um exemplo dessa chamada de função na Figura 12.16.

Processo de linkagem em uma hierarquia de herança

Na Seção 3.9, discutimos o processo de linkagem para criar um aplicativo GradeBook executável. Naquele exemplo, você viu que o código-objeto do cliente foi linkado com o código-objeto para classe GradeBook, bem como o código-objeto para qualquer classe da C++ Standard Library utilizada no código-cliente ou na classe GradeBook.

O processo de linkagem é semelhante a um programa que utiliza classes em uma hierarquia de herança. O processo requer o código-objeto para todas as classes utilizadas no programa e o código-objeto para as classes básicas diretas e indiretas de qualquer classe derivada utilizada pelo programa. Suponha que um cliente queira criar um aplicativo que utiliza a classe `BasePlusCommissionEmployee`, que é uma classe derivada de `CommissionEmployee` (veremos um exemplo disso na Seção 12.4.4). Ao compilar o aplicativo-cliente, o código-objeto do cliente deve ser linkado com o código-objeto das classes `BasePlusCommissionEmployee` e `CommissionEmployee`, porque `BasePlusCommissionEmployee` herda funções-membro de sua classe básica `CommissionEmployee`. O código também é linkado com o código-objeto de qualquer classe da C++ Standard Library utilizada na classe `CommissionEmployee`, na classe `BasePlusCommissionEmployee` ou no código-cliente. Isso fornece ao programa acesso às implementações de todas as funcionalidades que o programa pode utilizar.

12.4.4 Hierarquia de herança `CommissionEmployee–BasePlusCommissionEmployee` utilizando dados `protected`

Para permitir que a classe `BasePlusCommissionEmployee` acesse diretamente os membros de dados `CommissionEmployee` `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` e `commissionRate`, você pode declarar esses membros como `protected` na classe básica. Como discutimos na Seção 12.3, os membros `protected` de uma classe básica podem ser acessados por membros e `friends` da classe básica e por membros e `friends` de qualquer classe derivada dessa classe básica.



Boa prática de programação 12.1

Em primeiro lugar, declare os membros `public`, em segundo, os membros `protected` e, por último, os membros `private`.

Definindo a classe básica `CommissionEmployee` com dados `protected`

A classe `CommissionEmployee` (figuras 12.12–12.13) agora declara os membros de dados `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` e `commissionRate` como `protected` (Figura 12.12, linhas 33–37) em vez de `private`. As implementações de função-membro da Figura 12.13 são idênticas àsquelas da Figura 12.5.

```

1 // Figura 12.12: CommissionEmployee.h
2 // Definição da classe CommissionEmployee com dados protected.
3 #ifndef COMMISSION_H
4 #define COMMISSION_H
5
6 #include <string> // classe string padrão C++
7 using std::string;
8
9 class CommissionEmployee
10 {
11 public:
12     CommissionEmployee( const string &, const string &, const string &,
13                        double = 0.0, double = 0.0 );
14
15     void setFirstName( const string & ); // configura o nome
16     string getFirstName() const; // retorna o nome
17
18     void setLastName( const string & ); // configura o sobrenome
19     string getLastName() const; // retorna o sobrenome
20
21     void setSocialSecurityNumber( const string & ); // configura SSN

```

Figura 12.12 Definição da classe `CommissionEmployee` que declara dados `protected` para permitir acesso por classes derivadas.

(continua)


```

22     string getSocialSecurityNumber() const; // retorna SSN
23
24     void setGrossSales( double ); // configura a quantidade de vendas brutas
25     double getGrossSales() const; // retorna a quantidade de vendas brutas
26
27     void setCommissionRate( double ); // configura a taxa de comissão
28     double getCommissionRate() const; // retorna a taxa de comissão
29
30     double earnings() const; // calcula os rendimentos
31     void print() const; // imprime o objeto CommissionEmployee
32 protected:
33     string firstName;
34     string lastName;
35     string socialSecurityNumber;
36     double grossSales; // vendas brutas semanais
37     double commissionRate; // porcentagem da comissão
38 }; // fim da classe CommissionEmployee
39
40 #endif

```

Figura 12.12 Definição da classe `CommissionEmployee` que declara dados `protected` para permitir acesso por classes derivadas. (continuação)

```

1 // Figura 12.13: CommissionEmployee.cpp
2 // Definições de função-membro da classe CommissionEmployee.
3 #include <iostream>
4 using std::cout;
5
6 #include "CommissionEmployee.h" // definição da classe CommissionEmployee
7
8 // construtor
9 CommissionEmployee::CommissionEmployee(
10     const string &first, const string &last, const string &ssn,
11     double sales, double rate )
12 {
13     firstName = first; // deve validar
14     lastName = last; // deve validar
15     socialSecurityNumber = ssn; // deve validar
16     setGrossSales( sales ); // valida e armazena as vendas brutas
17     setCommissionRate( rate ); // valida e armazena a taxa de comissão
18 } // fim do construtor CommissionEmployee
19
20 // configura o nome
21 void CommissionEmployee::setFirstName( const string &first )
22 {
23     firstName = first; // deve validar
24 } // fim da função setFirstName
25
26 // retorna o nome
27 string CommissionEmployee::getFirstName() const
28 {
29     return firstName;
30 } // fim da função getFirstName
31
32 // configura o sobrenome

```

Figura 12.13 A classe `CommissionEmployee` com dados `protected`.

(continua)

```
33 void CommissionEmployee::setLastName( const string &last )
34 {
35     lastName = last; // deve validar
36 } // fim da função setLastName
37
38 // retorna o sobrenome
39 string CommissionEmployee::getLastName() const
40 {
41     return lastName;
42 } // fim da função getLastName
43
44 // configura o SSN
45 void CommissionEmployee::setSocialSecurityNumber( const string &ssn )
46 {
47     socialSecurityNumber = ssn; // deve validar
48 } // fim da função setSocialSecurityNumber
49
50 // retorna o SSN
51 string CommissionEmployee::getSocialSecurityNumber() const
52 {
53     return socialSecurityNumber;
54 } // fim da função getSocialSecurityNumber
55
56 // configura a quantidade de vendas brutas
57 void CommissionEmployee::setGrossSales( double sales )
58 {
59     grossSales = ( sales < 0.0 ) ? 0.0 : sales;
60 } // fim da função setGrossSales
61
62 // retorna a quantidade de vendas brutas
63 double CommissionEmployee::getGrossSales() const
64 {
65     return grossSales;
66 } // fim da função getGrossSales
67
68 // configura a taxa de comissão
69 void CommissionEmployee::setCommissionRate( double rate )
70 {
71     commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
72 } // fim da função setCommissionRate
73
74 // retorna a taxa de comissão
75 double CommissionEmployee::getCommissionRate() const
76 {
77     return commissionRate;
78 } // fim da função getCommissionRate
79
80 // calcula os rendimentos
81 double CommissionEmployee::earnings() const
82 {
83     return commissionRate * grossSales;
84 } // fim da função earnings
85
86 // imprime o objeto CommissionEmployee
87 void CommissionEmployee::print() const
```

Figura 12.13 A classe CommissionEmployee com dados protected.

(continua)

```

88 {
89     cout << "commission employee: " << firstName << ' ' << lastName
90         << "\nsocial security number: " << socialSecurityNumber
91         << "\ngross sales: " << grossSales
92         << "\ncommission rate: " << commissionRate;
93 } // fim da função print

```

Figura 12.13 A classe `CommissionEmployee` com dados `protected`.

(continuação)

Modificando a classe derivada *BasePlusCommissionEmployee*

Agora modificamos a classe `BasePlusCommissionEmployee` (figuras 12.14–12.15) para que ela herde da versão da classe `CommissionEmployee` nas figuras 12.12–12.13. Como a classe `BasePlusCommissionEmployee` herda a versão da classe `CommissionEmployee`, objetos da classe `BasePlusCommissionEmployee` podem acessar membros de dados herdados que são declarados `protected` na classe `CommissionEmployee` (isto é, membros de dados `firstName`, `lastName`, `socialSecurityNumber`, `grossSales` e `commissionRate`). Como resultado, o compilador não gera erros ao compilar as definições das funções-membro `BasePlusCommissionEmployee::earnings` e `print` na Figura 12.15 (linhas 32–36 e 39–47, respectivamente). Isso mostra os privilégios especiais que são concedidos a uma classe derivada para acessar membros de dados `protected` da classe básica. Os objetos de uma classe derivada também podem acessar membros `protected` em qualquer dessas classes básicas indiretas da classe derivada.

A classe `BasePlusCommissionEmployee` não herda o construtor da classe `CommissionEmployee`. Entretanto, o construtor da classe `BasePlusCommissionEmployee` (Figura 12.15, linhas 10–17) chama o construtor de `CommissionEmployee` explicitamente (linha 14). Lembre-se de que o construtor de `BasePlusCommissionEmployee` deve chamar explicitamente o construtor da classe `CommissionEmployee`, porque `CommissionEmployee` não contém um construtor-padrão que poderia ser invocado implicitamente.

```

1 // Figura 12.14: BasePlusCommissionEmployee.h
2 // Classe BasePlusCommissionEmployee derivada da classe
3 // CommissionEmployee.
4 #ifndef BASEPLUS_H
5 #define BASEPLUS_H
6
7 #include <string> // classe string padrão C++
8 using std::string;
9
10 #include "CommissionEmployee.h" // declaração da classe CommissionEmployee
11
12 class BasePlusCommissionEmployee : public CommissionEmployee
13 {
14 public:
15     BasePlusCommissionEmployee( const string &, const string &,
16                               const string &, double = 0.0, double = 0.0, double = 0.0 );
17
18     void setBaseSalary( double ); // configura o salário-base
19     double getBaseSalary() const; // retorna o salário-base
20
21     double earnings() const; // calcula os rendimentos
22     void print() const; // imprime o objeto BasePlusCommissionEmployee
23 private:
24     double baseSalary; // salário-base
25 }; // fim da classe BasePlusCommissionEmployee
26
27 #endif

```

Figura 12.14 Arquivo de cabeçalho da classe `BasePlusCommissionEmployee`.

```

1 // Figura 12.15: BasePlusCommissionEmployee.cpp
2 // Definições de função-membro da classe BasePlusCommissionEmployee.
3 #include <iostream>
4 using std::cout;
5
6 // definição da classe BasePlusCommissionEmployee
7 #include "BasePlusCommissionEmployee.h"
8
9 // construtor
10 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
11     const string &first, const string &last, const string &ssn,
12     double sales, double rate, double salary )
13     // chama explicitamente o construtor da classe básica
14     : CommissionEmployee( first, last, ssn, sales, rate )
15 {
16     setBaseSalary( salary ); // valida e armazena o salário-base
17 } // fim do construtor BasePlusCommissionEmployee
18
19 // configura o salário-base
20 void BasePlusCommissionEmployee::setBaseSalary( double salary )
21 {
22     baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
23 } // fim da função setBaseSalary
24
25 // retorna o salário-base
26 double BasePlusCommissionEmployee::getBaseSalary() const
27 {
28     return baseSalary;
29 } // fim da função getBaseSalary
30
31 // calcula os rendimentos
32 double BasePlusCommissionEmployee::earnings() const
33 {
34     // pode acessar dados protected da classe básica
35     return baseSalary + ( commissionRate * grossSales );
36 } // fim da função earnings
37
38 // imprime o objeto BasePlusCommissionEmployee
39 void BasePlusCommissionEmployee::print() const
40 {
41     // pode acessar dados protected da classe básica
42     cout << "base-salaried commission employee: " << firstName << ' '
43         << lastName << "\nsocial security number: " << socialSecurityNumber
44         << "\ngross sales: " << grossSales
45         << "\ncommission rate: " << commissionRate
46         << "\nbase salary: " << baseSalary;
47 } // fim da função print

```

Figura 12.15 O arquivo de implementação `BasePlusCommissionEmployee` para a classe `BasePlusCommissionEmployee` que herda dados `protected` de `CommissionEmployee`.

Testando a classe *BasePlusCommissionEmployee* modificada

A Figura 12.16 utiliza um objeto `BasePlusCommissionEmployee` para realizar as mesmas tarefas que a Figura 12.9 realizou em um objeto da primeira versão da classe `BasePlusCommissionEmployee` (figuras 12.7–12.8). Observe que as saídas dos dois programas são idênticas. Criamos a primeira classe `BasePlusCommissionEmployee` sem utilizar a herança e criamos essa versão de `BasePlusCommissionEmployee` utilizando a herança; entretanto, ambas as classes fornecem as mesmas funcionalidades. Observe que o código para a classe `BasePlusCommissionEmployee` (isto é, o cabeçalho e os arquivos de implementação), que tem 74 linhas, é considerável-

mente mais curto que o código para a versão da classe não herdada, que tem 154 linhas, porque a versão herdada absorve parte de suas funcionalidades a partir de `CommissionEmployee`, enquanto a versão não herdada não absorve nenhuma funcionalidade. Além disso, há agora apenas uma cópia da funcionalidade de `CommissionEmployee` declarada e definida na classe `CommissionEmployee`. Isso torna o código-fonte mais fácil de manter, modificar e depurar, porque o código-fonte relacionado com um `CommissionEmployee` existe somente nos arquivos das figuras 12.12–12.13.

Notas sobre a utilização dos dados *protected*

Neste exemplo, declaramos membros de dados da classe básica como *protected*, de modo que as classes derivadas pudessem modificar os dados diretamente. Herdar membros de dados *protected* aumenta ligeiramente o desempenho, porque podemos acessar os membros diretamente sem incorrer no overhead de chamadas a funções-membro *set* ou *get*. Na maioria dos casos, porém, é melhor utilizar os membros de dados *private* para incentivar a engenharia de software adequada e deixar as questões de otimização de código para o compilador. Seu código será mais fácil de manter, modificar e depurar.

```

1 // Figura 12.16: fig12_16.cpp
2 // Testando a classe BasePlusCommissionEmployee.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::fixed;
7
8 #include <iomanip>
9 using std::setprecision;
10
11 // definição da classe BasePlusCommissionEmployee
12 #include "BasePlusCommissionEmployee.h"
13
14 int main()
15 {
16     // instancia o objeto BasePlusCommissionEmployee
17     BasePlusCommissionEmployee
18         employee( "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
19
20     // configura a formatação de saída de ponto flutuante
21     cout << fixed << setprecision( 2 );
22
23     // obtém os dados de empregado comissionado
24     cout << "Employee information obtained by get functions: \n"
25         << "\nFirst name is " << employee.getFirstName()
26         << "\nLast name is " << employee.getLastName()
27         << "\nSocial security number is "
28         << employee.getSocialSecurityNumber()
29         << "\nGross sales is " << employee.getGrossSales()
30         << "\nCommission rate is " << employee.getCommissionRate()
31         << "\nBase salary is " << employee.getBaseSalary() << endl;
32
33     employee.setBaseSalary( 1000 ); // configura o salário-base
34
35     cout << "\nUpdated employee information output by print function: \n"
36         << endl;
37     employee.print(); // exibe as novas informações do empregado
38
39     // exibe os rendimentos do empregado
40     cout << "\n\nEmployee's earnings: $" << employee.earnings() << endl;
41
42     return 0;
43 } // fim de main

```

Figura 12.16 Os dados de classe básica *protected* podem ser acessados a partir da classe derivada.

(continua)

Employee information obtained by get functions:

```
First name is Bob
Last name is Lewis
Social security number is 333-33-3333
Gross sales is 5000.00
Commission rate is 0.04
Base salary is 300.00
```

Updated employee information output by print function:

```
base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 1000.00

Employee's earnings: $1200.00
```

Figura 12.16 Os dados de classe básica `protected` podem ser acessados a partir da classe derivada.

(continuação)

Utilizar membros de dados `protected` cria dois problemas importantes. Primeiro, o objeto de classe derivada não tem de utilizar uma função-membro para configurar o valor do membro de dados `protected` da classe básica. Portanto, um objeto de classe derivada pode atribuir facilmente um valor inválido ao membro de dados `protected`, deixando, assim, o objeto em um estado inconsistente. Por exemplo, com o membro de dados `grossSales` de `CommissionEmployee` declarado como `protected`, um objeto de classe derivada (por exemplo, `BasePlusCommissionEmployee`) pode atribuir um valor negativo a `grossSales`. O segundo problema com o uso de membros de dados `protected` é que é mais provável que as funções-membro de classe derivada sejam escritas de modo a depender da implementação da classe básica. Na prática, as classes derivadas só devem depender dos serviços da classe básica (isto é, funções-membro não-`private`), não da implementação da classe básica. Com membros de dados `protected` na classe básica, se a implementação de classe básica mudar, podemos precisar mudar todas as classes derivadas dessa classe básica. Por exemplo, se por alguma razão precisássemos mudar os nomes dos membros de dados `firstName` e `lastName` para `first` e `last`, então teríamos de fazer isso em todas as ocorrências em que uma classe derivada referencia diretamente esses membros de dados da classe básica. Em um caso como esse, diz-se que o software é **frágil** ou **quebradiço**, porque uma pequena alteração na classe básica pode ‘quebrar’ a implementação da classe derivada. O programador deve ser capaz de alterar a implementação da classe básica ao mesmo tempo em que ainda fornece os mesmos serviços para as classes derivadas. (Naturalmente, se os serviços de classe básica mudarem, devemos reimplementar nossas classes derivadas — um bom projeto orientado a objetos tenta evitar isso.)



Observação de engenharia de software 12.5

É apropriado utilizar o especificador de acesso `protected` quando uma classe básica tiver de fornecer um serviço (isto é, uma função-membro) apenas a suas classes derivadas (e *friends*), não a outros clientes.



Observação de engenharia de software 12.6

Declarar membros de dados de classe básica como `private` (em oposição a declará-los `protected`) permite aos programadores alterar a implementação da classe básica sem alterar as implementações de classe derivada.



Dica de prevenção de erro 12.1

Quando possível, evite incluir membros de dados `protected` em uma classe básica. Em vez disso, inclua funções-membro não-`private` que acessem membros de dados `private`, assegurando que o objeto mantenha um estado consistente.

12.4.5 Hierarquia de herança `CommissionEmployee`–`BasePlusCommissionEmployee` utilizando dados `private`

Agora reexamine nossa hierarquia mais uma vez, dessa vez utilizando práticas de engenharia de software melhores. A classe `CommissionEmployee` (figuras 12.17–12.18) agora declara membros de dados `firstName`, `lastName`, `socialSecurityNumber`, `grossSales`

e `commissionRate` como `private` (Figura 12.17, linhas 33–37) e fornece as funções-membro `public` `setFirstName`, `getFirstName`, `setLastName`, `getLastName`, `setSocialSecurityNumber`, `getSocialSecurityNumber`, `setGrossSales`, `getGrossSales`, `setCommissionRate`, `getCommissionRate`, `earnings` e `print` para manipular esses valores. Se decidirmos alterar o nome dos membros de dados, as definições `earnings` e `print` não exigirão modificação — somente as definições das funções-membro `get` e `set` que manipulam diretamente os membros de dados precisarão mudar. Observe que essas alterações ocorrem unicamente dentro da classe básica — não é necessária nenhuma mudança na classe derivada. Localizar os efeitos de alterações como esta é uma boa prática de engenharia de software. A classe derivada `BasePlusCommissionEmployee` (figuras 12.19–12.20) herda as funções-membro não-`private` de `CommissionEmployee` e pode acessar os membros `private` da classe básica via essas funções-membro.

Na implementação do construtor `CommissionEmployee` (Figura 12.18, linhas 9–16), observe que utilizamos inicializadores de membro (linha 12) para configurar os valores dos membros `firstName`, `lastName` e `socialSecurityNumber`. Mostramos como a classe derivada `BasePlusCommissionEmployee` (figuras 12.19–12.20) pode invocar funções-membro não-`private` da classe básica (`setFirstName`, `getFirstName`, `setLastName`, `getLastName`, `setSocialSecurityNumber` e `getSocialSecurityNumber`) para manipular esses membros de dados.

```

1 // Figura 12.17: CommissionEmployee.h
2 // Definição da classe CommissionEmployee com boa engenharia de software.
3 #ifndef COMMISSION_H
4 #define COMMISSION_H
5
6 #include <string> // classe string padrão C++
7 using std::string;
8
9 class CommissionEmployee
10 {
11 public:
12     CommissionEmployee( const string &, const string &, const string &,
13                        double = 0.0, double = 0.0 );
14
15     void setFirstName( const string & ); // configura o nome
16     string getFirstName() const; // retorna o nome
17
18     void setLastName( const string & ); // configura o sobrenome
19     string getLastName() const; // retorna o sobrenome
20
21     void setSocialSecurityNumber( const string & ); // configura o SSN
22     string getSocialSecurityNumber() const; // retorna o SSN
23
24     void setGrossSales( double ); // configura a quantidade de vendas brutas
25     double getGrossSales() const; // retorna a quantidade de vendas brutas
26
27     void setCommissionRate( double ); // configura a taxa de comissão
28     double getCommissionRate() const; // retorna a taxa de comissão
29
30     double earnings() const; // calcula os rendimentos
31     void print() const; // imprime o objeto CommissionEmployee
32 private:
33     string firstName;
34     string lastName;
35     string socialSecurityNumber;
36     double grossSales; // vendas brutas semanais
37     double commissionRate; // porcentagem da comissão
38 }; // fim da classe CommissionEmployee
39
40 #endif

```

Figura 12.17 Classe `CommissionEmployee` definida utilizando práticas de boa engenharia de software.


```

1 // Figura 12.18: CommissionEmployee.cpp
2 // Definições de função-membro da classe CommissionEmployee.
3 #include <iostream>
4 using std::cout;
5
6 #include "CommissionEmployee.h" // definição da classe CommissionEmployee
7
8 // construtor
9 CommissionEmployee::CommissionEmployee(
10     const string &first, const string &last, const string &ssn,
11     double sales, double rate )
12     : firstName( first ), lastName( last ), socialSecurityNumber( ssn )
13 {
14     setGrossSales( sales ); // valida e armazena as vendas brutas
15     setCommissionRate( rate ); // valida e armazena a taxa de comissão
16 } // fim do construtor CommissionEmployee
17
18 // configura o nome
19 void CommissionEmployee::setFirstName( const string &first )
20 {
21     firstName = first; // deve validar
22 } // fim da função setFirstName
23
24 // retorna o nome
25 string CommissionEmployee::getFirstName() const
26 {
27     return firstName;
28 } // fim da função getFirstName
29
30 // configura o sobrenome
31 void CommissionEmployee::setLastName( const string &last )
32 {
33     lastName = last; // deve validar
34 } // fim da função setLastName
35
36 // retorna o sobrenome
37 string CommissionEmployee::getLastName() const
38 {
39     return lastName;
40 } // fim da função getLastName
41
42 // configura o SSN
43 void CommissionEmployee::setSocialSecurityNumber( const string &ssn )
44 {
45     socialSecurityNumber = ssn; // deve validar
46 } // fim da função setSocialSecurityNumber
47
48 // retorna o SSN
49 string CommissionEmployee::getSocialSecurityNumber() const
50 {
51     return socialSecurityNumber;
52 } // fim da função getSocialSecurityNumber
53
54 // configura a quantidade de vendas brutas
55 void CommissionEmployee::setGrossSales( double sales )
56 {

```

Figura 12.18 Arquivo de implementação da classe `CommissionEmployee`: a classe `CommissionEmployee` utiliza funções-membro para manipular seus dados `private`. (continua)

```

57     grossSales = ( sales < 0.0 ) ? 0.0 : sales;
58 } // fim da função setGrossSales
59
60 // retorna a quantidade de vendas brutas
61 double CommissionEmployee::getGrossSales() const
62 {
63     return grossSales;
64 } // fim da função getGrossSales
65
66 // configura a taxa de comissão
67 void CommissionEmployee::setCommissionRate( double rate )
68 {
69     commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
70 } // fim da função setCommissionRate
71
72 // retorna a taxa de comissão
73 double CommissionEmployee::getCommissionRate() const
74 {
75     return commissionRate;
76 } // fim da função getCommissionRate
77
78 // calcula os rendimentos
79 double CommissionEmployee::earnings() const
80 {
81     return getCommissionRate() * getGrossSales();
82 } // fim da função earnings
83
84 // imprime o objeto CommissionEmployee
85 void CommissionEmployee::print() const
86 {
87     cout << "commission employee: "
88         << getFirstName() << ' ' << getLastName()
89         << "\nsocial security number: " << getSocialSecurityNumber()
90         << "\ngross sales: " << getGrossSales()
91         << "\ncommission rate: " << getCommissionRate();
92 } // fim da função print

```

Figura 12.18 Arquivo de implementação da classe `CommissionEmployee`: a classe `CommissionEmployee` utiliza funções-membro para manipular seus dados `private`. (continuação)



Dica de desempenho 12.2

Utilizar uma função-membro para acessar o valor de um membro de dados é ligeiramente mais lento que acessar os dados diretamente. Entretanto, os compiladores de otimização atuais são projetados com cuidado para realizar muitas otimizações implicitamente (como colocar inline chamadas de função-membro `get` e `set`). Como resultado, os programadores devem escrever código que obedeça aos princípios apropriados da engenharia de software e deixar questões de otimização para o compilador. Uma boa regra é ‘Não se antecipe ao compilador’.

A classe `BasePlusCommissionEmployee` (figuras 12.19–12.20) tem várias alterações em suas implementações de função-membro (Figura 12.20) que a distinguem da versão anterior da classe (figuras 12.14–12.15). As funções-membro `earnings` (Figura 12.20, linhas 32–35) e `print` (linhas 38–46) invocam a função-membro `getBaseSalary` para obter o valor do salário-base, em vez de acessar `baseSalary` diretamente. Isso isola `earnings` e `print` de potenciais alterações na implementação do membro de dados `baseSalary`. Por exemplo, se decidirmos renomear o membro de dados `baseSalary` ou alterar seu tipo, somente as funções-membro `setBaseSalary` e `getBaseSalary` precisarão mudar.

A função `earnings` da classe `BasePlusCommissionEmployee` (Figura 12.20, linhas 32–35) redefine a função-membro `earnings` da classe `CommissionEmployee` (Figura 12.18, linhas 79–82) para calcular os rendimentos de um empregado comissionado com salário-base. A versão de `earnings` da classe `BasePlusCommissionEmployee` obtém a parte dos rendimentos do empregado baseada exclusivamente na comissão chamando a função `earnings` da classe básica `CommissionEmployee` com a expressão `CommissionEmployee::earnings()`

(Figura 12.20, linha 34). A função `earnings` de `BasePlusCommissionEmployee` então adiciona o salário-base a esse valor para calcular os rendimentos totais do empregado. Observe a sintaxe utilizada para invocar uma função-membro da classe básica redefinida a partir de uma classe derivada — coloque o nome da classe básica e o operador de resolução de escopo binário (`::`) antes do nome da função-membro da classe básica. Essa invocação de função-membro é uma boa prática de engenharia de software: considerando a “Observação de engenharia de software 9.9”, lembre-se de que, se a função-membro de um objeto realiza as ações necessárias para outro objeto, devemos chamar essa função-membro em vez de duplicar o corpo do seu código. Fazendo a função `earnings` de `BasePlusCommissionEmployee` invocar a função `earnings` de `CommissionEmployee` para calcular parte dos rendimentos de um objeto `BasePlusCommissionEmployee`, evitamos duplicar o código e reduzimos problemas de manutenção de código.

```

1 // Figura 12.19: BasePlusCommissionEmployee.h
2 // Classe BasePlusCommissionEmployee derivada da classe
3 // CommissionEmployee.
4 #ifndef BASEPLUS_H
5 #define BASEPLUS_H
6
7 #include <string> // classe string padrão C++
8 using std::string;
9
10 #include "CommissionEmployee.h" // declaração da classe CommissionEmployee
11
12 class BasePlusCommissionEmployee : public CommissionEmployee
13 {
14 public:
15     BasePlusCommissionEmployee( const string &, const string &,
16                               const string &, double = 0.0, double = 0.0, double = 0.0 );
17
18     void setBaseSalary( double ); // configura o salário-base
19     double getBaseSalary() const; // retorna o salário-base
20
21     double earnings() const; // calcula os rendimentos
22     void print() const; // imprime o objeto BasePlusCommissionEmployee
23 private:
24     double baseSalary; // salário-base
25 }; // fim da classe BasePlusCommissionEmployee
26
27 #endif

```

Figura 12.19 Arquivo de cabeçalho da classe `BasePlusCommissionEmployee`.

```

1 // Figura 12.20: BasePlusCommissionEmployee.cpp
2 // Definições de função-membro da classe BasePlusCommissionEmployee.
3 #include <iostream>
4 using std::cout;
5
6 // definição da classe BasePlusCommissionEmployee
7 #include "BasePlusCommissionEmployee.h"
8
9 // construtor
10 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
11     const string &first, const string &last, const string &ssn,
12     double sales, double rate, double salary )
13     // chama explicitamente o construtor da classe básica

```

Figura 12.20 A classe `BasePlusCommissionEmployee` que herda da classe `CommissionEmployee` mas não pode acessar diretamente os dados `private` da classe. (continua)

```

14 : CommissionEmployee( first, last, ssn, sales, rate )
15 {
16     setBaseSalary( salary ); // valida e armazena salário-base
17 } // fim do construtor BasePlusCommissionEmployee
18
19 // configura o salário-base
20 void BasePlusCommissionEmployee::setBaseSalary( double salary )
21 {
22     baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
23 } // fim da função setBaseSalary
24
25 // retorna o salário-base
26 double BasePlusCommissionEmployee::getBaseSalary() const
27 {
28     return baseSalary;
29 } // fim da função getBaseSalary
30
31 // calcula os rendimentos
32 double BasePlusCommissionEmployee::earnings() const
33 {
34     return getBaseSalary() + CommissionEmployee::earnings();
35 } // fim da função earnings
36
37 // imprime o objeto BasePlusCommissionEmployee
38 void BasePlusCommissionEmployee::print() const
39 {
40     cout << "base-salaried ";
41
42     // invoca a função print de CommissionEmployee
43     CommissionEmployee::print();
44
45     cout << "\nbase salary: " << getBaseSalary();
46 } // fim da função print

```

Figura 12.20 A classe `BasePlusCommissionEmployee` que herda da classe `CommissionEmployee` mas não pode acessar diretamente os dados `private` da classe. (continuação)



Erro comum de programação 12.2

Quando uma função-membro de classe básica é redefinida para uma classe derivada, a versão da classe derivada frequentemente chama a versão da classe básica para fazer o trabalho adicional. A falha em utilizar o operador `::` prefixado com o nome da classe básica ao referenciar a função-membro da classe básica causa a recursão infinita, porque a função-membro da classe derivada chamaria a si própria.



Erro comum de programação 12.3

Incluir uma função-membro de classe básica com uma assinatura diferente na classe derivada oculta a versão de classe básica da função. Tentativas de chamar a versão de classe básica pela interface `public` de um objeto de classe derivada resultam em erros de compilação.

De maneira semelhante, a função `print` de `BasePlusCommissionEmployee` (Figura 12.20, linhas 38–46) redefine a função-membro `print` da classe `CommissionEmployee` (Figura 12.18, linhas 85–92) para gerar a saída de informações que sejam apropriadas para o empregado comissionado com salário-base. A versão da classe `BasePlusCommissionEmployee` exibe parte das informações de um objeto `BasePlusCommissionEmployee` (isto é, a string "commission employee" e os valores dos membros de dados `private` da classe `CommissionEmployee`) chamando a função-membro `print` de `CommissionEmployee` com o nome qualificado `CommissionEmployee::print()` (Figura 12.20, linha 43). A função `print` de `BasePlusCommissionEmployee` então gera saída das informações restantes do objeto `BasePlusCommissionEmployee` (isto é, o valor do salário-base da classe `BasePlusCommissionEmployee`).

A Figura 12.21 realiza as mesmas manipulações em um objeto `BasePlusCommissionEmployee` que as figuras 12.9 e 12.16 sobre os objetos das classes `CommissionEmployee` e `BasePlusCommissionEmployee`, respectivamente. Embora toda a classe ‘empregado comissionado com salário-base’ comporte-se de modo idêntico, a classe `BasePlusCommissionEmployee` tem o melhor projeto. Utilizando a herança e chamando as funções-membro que ocultam os dados e asseguram a consistência, criamos eficiente e efetivamente uma classe bem-projetada.

Nesta seção, você viu um conjunto evolutivo de exemplos que foi cuidadosamente projetado para ensinar as capacidades-chave para a boa engenharia de software com herança. Você aprendeu a criar uma classe derivada usando herança, a utilizar membros de classe básica `protected` para permitir que uma classe derivada acesse membros de dados da classe básica herdados e a redefinir as funções de classe básica para fornecer versões mais apropriadas aos objetos de classe derivada. Além disso, você aprendeu a aplicar as técnicas de engenharia de software apresentadas nos capítulos 9–10 e neste capítulo para criar classes que são fáceis de manter, modificar e depurar.

```

1  // Figura 12.21: fig12_21.cpp
2  // Testando a classe BasePlusCommissionEmployee.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6  using std::fixed;
7
8  #include <iomanip>
9  using std::setprecision;
10
11 // definição da classe BasePlusCommissionEmployee
12 #include "BasePlusCommissionEmployee.h"
13
14 int main()
15 {
16     // instancia o objeto BasePlusCommissionEmployee
17     BasePlusCommissionEmployee
18         employee( "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
19
20     // configura a formatação de saída de ponto flutuante
21     cout << fixed << setprecision( 2 );
22
23     // obtém os dados do empregado comissionado
24     cout << "Employee information obtained by get functions: \n"
25         << "\nFirst name is " << employee.getFirstName()
26         << "\nLast name is " << employee.getLastName()
27         << "\nSocial security number is "
28         << employee.getSocialSecurityNumber()
29         << "\nGross sales is " << employee.getGrossSales()
30         << "\nCommission rate is " << employee.getCommissionRate()
31         << "\nBase salary is " << employee.getBaseSalary() << endl;
32
33     employee.setBaseSalary( 1000 ); // configura o salário-base
34
35     cout << "\nUpdated employee information output by print function: \n"
36         << endl;
37     employee.print(); // exibe as novas informações do empregado
38
39     // exibe os rendimentos do empregado
40     cout << "\n\nEmployee's earnings: $" << employee.earnings() << endl;
41
42     return 0;
43 } // fim de main

```

Figura 12.21 Dados `private` de classe básica são acessíveis a uma classe derivada via função-membro `public` ou `protected` herdada pela classe derivada. (continua)

```

Employee information obtained by get functions:

First name is Bob
Last name is Lewis
Social security number is 333-33-3333
Gross sales is 5000.00
Commission rate is 0.04
Base salary is 300.00

Updated employee information output by print function:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 1000.00

Employee's earnings: $1200.00

```

Figura 12.21 Dados private de classe básica são acessíveis a uma classe derivada via função-membro public ou protected herdada pela classe derivada. (continuação)

12.5 Construtores e destrutores em classes derivadas

Como explicamos na seção anterior, instanciar um objeto de classe derivada inicia uma cadeia de chamadas de construtor em que o construtor de classe derivada, antes de realizar suas próprias tarefas, invoca o construtor da sua classe básica direta explicitamente (via um inicializador de membro de classe básica) ou implicitamente (chamando o construtor-padrão da classe básica). De maneira semelhante, se a classe básica é derivada de outra classe, o construtor da classe básica é solicitado a invocar o construtor da próxima classe na hierarquia e assim por diante. O último construtor chamado nessa cadeia é o construtor da classe na base da hierarquia, cujo corpo na realidade termina de executar primeiro. O corpo do construtor da classe derivada original é o último a terminar de executar. Cada construtor de classe básica inicializa os membros de dados da classe básica que o objeto da classe derivada herda. Por exemplo, considere a hierarquia `CommissionEmployee/BasePlusCommissionEmployee` das figuras 12.17–12.20. Quando um programa cria um objeto da classe `BasePlusCommissionEmployee`, o construtor `CommissionEmployee` é chamado. Visto que a classe `CommissionEmployee` está na base da hierarquia, seu construtor executa, inicializando os membros de dados private de `CommissionEmployee` que fazem parte do objeto `BasePlusCommissionEmployee`. Quando o construtor de `CommissionEmployee` termina de executar, ele retorna o controle para o construtor de `BasePlusCommissionEmployee`, o qual inicializa o `baseSalary` do objeto `BasePlusCommissionEmployee`.



Observação de engenharia de software 12.7

Quando um programa cria um objeto de classe derivada, o construtor da classe derivada chama imediatamente o construtor da classe básica, o corpo do construtor da classe básica executa, em seguida, os inicializadores de membro da classe derivada executam e, por fim, o corpo do construtor da classe derivada executa. Esse processo coloca a hierarquia em cascata se ela contiver mais de dois níveis.

Quando um objeto de classe derivada é destruído, o programa chama o destrutor desse objeto. Isso inicia uma cadeia (ou cascata) de chamadas de destrutor em que o destrutor da classe derivada, os destrutores das classes básicas diretas e indiretas, e os membros das classes executam na ordem inversa em que os construtores executaram. Quando o destrutor de um objeto de classe derivada é chamado, o destrutor realiza sua tarefa, então invoca o destrutor da próxima classe básica na hierarquia. Esse processo se repete até que o destrutor da classe básica final na parte superior da hierarquia é chamado. Então o objeto é removido da memória.



Observação de engenharia de software 12.8

Suponha que tivéssemos criado um objeto de uma classe derivada em que tanto a classe básica como a classe derivada contivessem objetos de outras classes. Quando um objeto dessa classe derivada é criado, os construtores para os objetos-membro da classe básica executam primeiro, em seguida, o construtor da classe básica, os construtores para os objetos de membro da classe derivada e o construtor da classe derivada executam, nessa ordem. Os destrutores para objetos de classe derivada são chamados na ordem inversa de seus construtores correspondentes.

Os construtores, destrutores e operadores de atribuição sobrecarregados de uma classe básica (ver Capítulo 11, “Sobrecarga dos operadores; objetos string e array”) não são herdados por classes derivadas. Entretanto, os construtores, destrutores e operadores de atribuição sobrecarregados de uma classe derivada podem chamar construtores, destrutores e operadores de atribuição sobrecarregados da classe básica.

Nosso próximo exemplo revisita a hierarquia de empregados comissionados definindo a classe `CommissionEmployee` (figuras 12.22–12.23) e a classe `BasePlusCommissionEmployee` (figuras 12.24–12.25) que contêm construtores e destrutores, cada um dos quais imprime uma mensagem quando é invocado. Como você verá na saída da Figura 12.26, essas mensagens demonstram a ordem em que os construtores e destrutores são chamados por objetos em uma hierarquia de herança.

```

1  // Figura 12.22: CommissionEmployee.h
2  // Classe CommissionEmployee representa um empregado comissionado.
3  #ifndef COMMISSION_H
4  #define COMMISSION_H
5
6  #include <string> // classe string padrão C++
7  using std::string;
8
9  class CommissionEmployee
10 {
11 public:
12     CommissionEmployee( const string &, const string &, const string &,
13                        double = 0.0, double = 0.0 );
14     ~CommissionEmployee(); // destrutor
15
16     void setFirstName( const string & ); // configura o nome
17     string getFirstName() const; // retorna o nome
18
19     void setLastName( const string & ); // configura o sobrenome
20     string getLastName() const; // retorna o sobrenome
21
22     void setSocialSecurityNumber( const string & ); // configura o SSN
23     string getSocialSecurityNumber() const; // retorna o SSN
24
25     void setGrossSales( double ); // configura a quantidade de vendas brutas
26     double getGrossSales() const; // retorna a quantidade de vendas brutas
27
28     void setCommissionRate( double ); // configura a taxa de comissão
29     double getCommissionRate() const; // retorna a taxa de comissão
30
31     double earnings() const; // calcula os rendimentos
32     void print() const; // imprime o objeto CommissionEmployee
33 private:
34     string firstName;
35     string lastName;
36     string socialSecurityNumber;
37     double grossSales; // vendas brutas semanais
38     double commissionRate; // porcentagem da comissão
39 }; // fim da classe CommissionEmployee
40
41 #endif

```

Figura 12.22 Arquivo de cabeçalho da classe `CommissionEmployee`.

Neste exemplo, modificamos o construtor `CommissionEmployee` (linhas 10–21 da Figura 12.23) e incluímos um destrutor `CommissionEmployee` (linhas 24–29), cada um dos quais gera saída de uma linha de texto sobre sua invocação. Também modificamos o construtor `BasePlusCommissionEmployee` (linhas 11–22 da Figura 12.25) e incluímos um destrutor `BasePlusCommissionEmployee` (linhas 25–30), cada um dos quais gera saída de uma linha de texto na sua invocação.

```

1 // Figura 12.23: CommissionEmployee.cpp
2 // Definições de função-membro da classe CommissionEmployee.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "CommissionEmployee.h" // definição da classe CommissionEmployee
8
9 // construtor
10 CommissionEmployee::CommissionEmployee(
11     const string &first, const string &last, const string &ssn,
12     double sales, double rate )
13     : firstName( first ), lastName( last ), socialSecurityNumber( ssn )
14 {
15     setGrossSales( sales ); // valida e armazena as vendas brutas
16     setCommissionRate( rate ); // valida e armazena a taxa de comissão
17
18     cout << "CommissionEmployee constructor: " << endl;
19     print();
20     cout << "\n\n";
21 } // fim do construtor CommissionEmployee
22
23 // destrutor
24 CommissionEmployee::~CommissionEmployee()
25 {
26     cout << "CommissionEmployee destructor: " << endl;
27     print();
28     cout << "\n\n";
29 } // fim do destrutor CommissionEmployee
30
31 // configura o nome
32 void CommissionEmployee::setFirstName( const string &first )
33 {
34     firstName = first; // deve validar
35 } // fim da função setFirstName
36
37 // retorna o nome
38 string CommissionEmployee::getFirstName() const
39 {
40     return firstName;
41 } // fim da função getFirstName
42
43 // configura o sobrenome
44 void CommissionEmployee::setLastName( const string &last )
45 {
46     lastName = last; // deve validar
47 } // fim da função setLastName
48
49 // retorna o sobrenome
50 string CommissionEmployee::getLastName() const

```

Figura 12.23 O construtor de `CommissionEmployee` gera saída de texto.

(continua)

```

51 {
52     return lastName;
53 } // fim da função getLastName
54
55 // configura o SSN
56 void CommissionEmployee::setSocialSecurityNumber( const string &ssn )
57 {
58     socialSecurityNumber = ssn; // deve validar
59 } // fim da função setSocialSecurityNumber
60
61 // retorna o SSN
62 string CommissionEmployee::getSocialSecurityNumber() const
63 {
64     return socialSecurityNumber;
65 } // fim da função getSocialSecurityNumber
66
67 // configura a quantidade de vendas brutas
68 void CommissionEmployee::setGrossSales( double sales )
69 {
70     grossSales = ( sales < 0.0 ) ? 0.0 : sales;
71 } // fim da função setGrossSales
72
73 // retorna a quantidade de vendas brutas
74 double CommissionEmployee::getGrossSales() const
75 {
76     return grossSales;
77 } // fim da função getGrossSales
78
79 // configura a taxa de comissão
80 void CommissionEmployee::setCommissionRate( double rate )
81 {
82     commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
83 } // fim da função setCommissionRate
84
85 // retorna a taxa de comissão
86 double CommissionEmployee::getCommissionRate() const
87 {
88     return commissionRate;
89 } // fim da função getCommissionRate
90
91 // calcula os rendimentos
92 double CommissionEmployee::earnings() const
93 {
94     return getCommissionRate() * getGrossSales();
95 } // fim da função earnings
96
97 // imprime o objeto CommissionEmployee
98 void CommissionEmployee::print() const
99 {
100     cout << "commission employee: "
101         << getFirstName() << ' ' << getLastName()
102         << "\nsocial security number: " << getSocialSecurityNumber()
103         << "\ngross sales: " << getGrossSales()
104         << "\ncommission rate: " << getCommissionRate();
105 } // fim da função print

```

Figura 12.23 O construtor de `CommissionEmployee` gera saída de texto.

(continuação)

```

1 // Figura 12.24: BasePlusCommissionEmployee.h
2 // Classe BasePlusCommissionEmployee derivada da classe
3 // CommissionEmployee.
4 #ifndef BASEPLUS_H
5 #define BASEPLUS_H
6
7 #include <string> // classe string padrão C++
8 using std::string;
9
10 #include "CommissionEmployee.h" // declaração da classe CommissionEmployee
11
12 class BasePlusCommissionEmployee : public CommissionEmployee
13 {
14 public:
15     BasePlusCommissionEmployee( const string &, const string &,
16                               const string &, double = 0.0, double = 0.0, double = 0.0 );
17     ~BasePlusCommissionEmployee(); // destrutor
18
19     void setBaseSalary( double ); // configura o salário-base
20     double getBaseSalary() const; // retorna o salário-base
21
22     double earnings() const; // calcula os rendimentos
23     void print() const; // imprime o objeto BasePlusCommissionEmployee
24 private:
25     double baseSalary; // salário-base
26 }; // fim da classe BasePlusCommissionEmployee
27
28 #endif

```

Figura 12.24 Arquivo de cabeçalho da classe BasePlusCommissionEmployee.

```

1 // Figura 12.25: BasePlusCommissionEmployee.cpp
2 // Definições de funções-membro da classe BasePlusCommissionEmployee.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 // definição da classe BasePlusCommissionEmployee
8 #include "BasePlusCommissionEmployee.h"
9
10 // construtor
11 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
12     const string &first, const string &last, const string &ssn,
13     double sales, double rate, double salary )
14     // chama explicitamente o construtor da classe básica
15     : CommissionEmployee( first, last, ssn, sales, rate )
16 {
17     setBaseSalary( salary ); // valida e armazena salário-base
18
19     cout << "BasePlusCommissionEmployee constructor: " << endl;
20     print();
21     cout << "\n\n";
22 } // fim do construtor BasePlusCommissionEmployee
23

```

Figura 12.25 O construtor de BasePlusCommissionEmployee gera saída de texto.

(continua)

```

24 // destrutor
25 BasePlusCommissionEmployee::~BasePlusCommissionEmployee()
26 {
27     cout << "BasePlusCommissionEmployee destructor: " << endl;
28     print();
29     cout << "\n\n";
30 } // fim do destrutor BasePlusCommissionEmployee
31
32 // configura o salário-base
33 void BasePlusCommissionEmployee::setBaseSalary( double salary )
34 {
35     baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
36 } // fim da função setBaseSalary
37
38 // retorna o salário-base
39 double BasePlusCommissionEmployee::getBaseSalary() const
40 {
41     return baseSalary;
42 } // fim da função getBaseSalary
43
44 // calcula os rendimentos
45 double BasePlusCommissionEmployee::earnings() const
46 {
47     return getBaseSalary() + CommissionEmployee::earnings();
48 } // fim da função earnings
49
50 // imprime o objeto BasePlusCommissionEmployee
51 void BasePlusCommissionEmployee::print() const
52 {
53     cout << "base-salaried ";
54
55     // invoca a função print de CommissionEmployee
56     CommissionEmployee::print();
57
58     cout << "\nbase salary: " << getBaseSalary();
59 } // fim da função print

```

Figura 12.25 O construtor de BasePlusCommissionEmployee gera saída de texto.

(continuação)

A Figura 12.26 demonstra a ordem em que construtores e destrutores são chamados para objetos de classes que fazem parte de uma hierarquia de herança. A função `main` (linhas 15–34) começa instanciando o objeto `CommissionEmployee employee1` (linhas 21–22) em um bloco separado dentro de `main` (linhas 20–23). O objeto entra e sai imediatamente do escopo (o fim do bloco é alcançado logo que o objeto é criado), então o construtor e o destrutor `CommissionEmployee` são chamados. Em seguida, as linhas 26–27 instanciam o objeto `BasePlusCommissionEmployee employee2`. Isso invoca o construtor `CommissionEmployee` para exibir as saídas com os valores passados a partir do construtor `BasePlusCommissionEmployee`, então a saída especificada no construtor `BasePlusCommissionEmployee` é realizada. As linhas 30–31 instanciam então o objeto `BasePlusCommissionEmployee employee3`. Novamente, os construtores `CommissionEmployee` e `BasePlusCommissionEmployee` são ambos chamados. Observe que, em cada caso, o corpo do construtor `CommissionEmployee` é executado antes de o corpo do construtor `BasePlusCommissionEmployee` executar. Quando o fim de `main` é alcançado, os destrutores são chamados pelos objetos `employee2` e `employee3`. Mas, como os destrutores são chamados na ordem inversa de seus construtores correspondentes, o destrutor `BasePlusCommissionEmployee` e o destrutor `CommissionEmployee` são chamados (nessa ordem) para o objeto `employee3` e, em seguida, os destrutores `BasePlusCommissionEmployee` e `CommissionEmployee` são chamados (nessa ordem) para o objeto `employee2`.

```

1 // Figura 12.26: fig12_26.cpp
2 // Ordem de exibição em que a classe básica e construtores e destrutores
3 // da classe derivada são chamados.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7 using std::fixed;
8
9 #include <iomanip>
10 using std::setprecision;
11
12 // definição da classe BasePlusCommissionEmployee
13 #include "BasePlusCommissionEmployee.h"
14
15 int main()
16 {
17     // configura a formatação de saída de ponto flutuante
18     cout << fixed << setprecision( 2 );
19
20     { // inicia novo escopo
21         CommissionEmployee employee1(
22             "Bob", "Lewis", "333-33-3333", 5000, .04 );
23     } // termina o escopo
24
25     cout << endl;
26     BasePlusCommissionEmployee
27         employee2( "Lisa", "Jones", "555-55-5555", 2000, .06, 800 );
28
29     cout << endl;
30     BasePlusCommissionEmployee
31         employee3( "Mark", "Sands", "888-88-8888", 8000, .15, 2000 );
32     cout << endl;
33     return 0;
34 } // fim de main

```

CommissionEmployee constructor:
 commission employee: Bob Lewis
 social security number: 333-33-3333
 gross sales: 5000.00
 commission rate: 0.04

CommissionEmployee destructor:
 commission employee: Bob Lewis
 social security number: 333-33-3333
 gross sales: 5000.00
 commission rate: 0.04

CommissionEmployee constructor:
 base-salaried commission employee: Lisa Jones
 social security number: 555-55-5555
 gross sales: 2000.00
 commission rate: 0.06

BasePlusCommissionEmployee constructor:
 base-salaried commission employee: Lisa Jones
 social security number: 555-55-5555

Figura 12.26 Ordem de chamada do construtor e do destrutor.

(continua)

```

gross sales: 2000.00
commission rate: 0.06
base salary: 800.00

CommissionEmployee constructor:
commission employee: Mark Sands
social security number: 888-88-8888
gross sales: 8000.00
commission rate: 0.15

BasePlusCommissionEmployee constructor:
base-salaried commission employee: Mark Sands
social security number: 888-88-8888
gross sales: 8000.00
commission rate: 0.15
base salary: 2000.00

BasePlusCommissionEmployee destructor:
base-salaried commission employee: Mark Sands
social security number: 888-88-8888
gross sales: 8000.00
commission rate: 0.15
base salary: 2000.00

CommissionEmployee destructor:
commission employee: Mark Sands
social security number: 888-88-8888
gross sales: 8000.00
commission rate: 0.15

BasePlusCommissionEmployee destructor:
base-salaried commission employee: Lisa Jones
social security number: 555-55-5555
gross sales: 2000.00
commission rate: 0.06
base salary: 800.00

CommissionEmployee destructor:
commission employee: Lisa Jones
social security number: 555-55-5555
gross sales: 2000.00
commission rate: 0.06

```

Figura 12.26 Ordem de chamada do construtor e do destrutor.

(continuação)

12.6 Herança public, protected e private

Ao derivar uma classe de uma classe básica, a classe básica pode ser herdada pela herança `public`, `protected` ou `private`. O uso de herança `protected` e `private` é raro, e cada uma delas só deve ser usada com grande cuidado; normalmente, utilizamos a herança `public` neste livro. (O Capítulo 21 demonstra a herança `private` como uma alternativa à composição.) A Figura 12.27 resume para cada tipo de herança a acessibilidade de membros de classe básica em uma classe derivada. A primeira coluna contém os especificadores de acesso de classe básica.

Ao derivar uma classe de uma classe básica `public`, os membros `public` da classe básica se tornam membros `public` da classe derivada e os membros `protected` da classe básica se tornam os membros `protected` da classe derivada. Os membros `private` de uma classe básica nunca são acessíveis diretamente de uma classe derivada, mas podem ser acessados por meio das chamadas aos membros `public` e `protected` da classe básica.

Especificador de acesso de membro de classe básica	Tipo de herança		
	Herança public	Herança protected	Herança private
public	public na classe derivada. Pode ser acessada diretamente por funções-membro, funções friend e funções não-membro.	protected na classe derivada. Pode ser acessada diretamente por funções-membro e funções friend.	private na classe derivada. Pode ser acessada diretamente por funções-membro e funções friend.
protected	protected na classe derivada. Pode ser acessada diretamente por funções-membro e funções friend.	protected na classe derivada. Pode ser acessada diretamente por funções-membro e funções friend.	private na classe derivada. Pode ser acessada diretamente por funções-membro e funções friend.
private	Oculto na classe derivada. Pode ser acessada por funções-membro e funções friend por meio das funções-membro public ou protected da classe básica.	Oculto na classe derivada. Pode ser acessada por funções-membro e funções friend por meio das funções-membro public ou protected da classe básica.	Oculto na classe derivada. Pode ser acessada por funções-membro e funções friend por meio das funções-membro public ou protected da classe básica.

Figura 12.27 Resumo da acessibilidade do membro de classe básica em uma classe derivada.

Ao derivar de uma classe básica `protected`, os membros `public` e `protected` da classe básica tornam-se os membros `protected` da classe derivada. Ao derivar de uma classe básica `private`, os membros `public` e `protected` da classe básica tornam-se os membros `private` (por exemplo, as funções tornam-se funções utilitárias) da classe derivada. As heranças `private` e `protected` não são relacionamentos *é um*.

12.7 Engenharia de software com herança

Nesta seção, discutimos o uso de herança para personalizar o software existente. Quando utilizamos herança para criar uma nova classe a partir de uma classe existente, a nova classe herda os membros de dados e funções-membro da classe existente, como descrito na Figura 12.27. Podemos personalizar a nova classe para atender às nossas necessidades incluindo membros adicionais e redefinindo membros de classe básica. O programador de classe derivada faz isso em C++ sem acessar o código-fonte da classe básica. A classe derivada deve ser capaz de se linkar ao código-objeto da classe básica. Essa poderosa capacidade é atraente para fornecedores de softwares independentes (*independent software vendors* – ISV). Os ISV podem desenvolver classes ‘proprietárias’ (patenteadas) para venda ou licenciamento e disponibilizá-las para usuários no formato de código-objeto. Os usuários podem então derivar novas classes dessas classes de biblioteca rapidamente e sem acessar o código-fonte proprietário (patenteado) dos ISV. Tudo o que os ISV precisam fornecer ao código-objeto são os arquivos de cabeçalho.

Às vezes é difícil para os alunos avaliarem o escopo de problemas enfrentados por projetistas que trabalham em projetos de software de larga escala na indústria. Pessoas experientes nesses projetos dizem que a reutilização efetiva de software melhora o processo de desenvolvimento de software. A programação orientada a objetos facilita a reutilização de software, diminuindo, assim, as horas de desenvolvimento e aprimorando a qualidade do software.

A disponibilidade de bibliotecas de classe substanciais e úteis fornece os benefícios máximos de reutilização de software por meio da herança. Assim como o software comercial produzido por fornecedores de software independentes tornou-se uma indústria de crescimento explosivo com a chegada do computador pessoal, o interesse pela criação e venda de bibliotecas de classes também está crescendo exponencialmente. Os projetistas de aplicativo constroem seus aplicativos com essas bibliotecas e projetistas de biblioteca estão sendo recompensados por ter suas bibliotecas incluídas nos aplicativos. As bibliotecas C++ padrão que vêm com compiladores C++ tendem a ser de uso mais geral e limitado em escopo. Entretanto, há um compromisso mundial maciço para o desenvolvimento de bibliotecas de classes para uma variedade enorme de áreas de aplicação.



Observação de engenharia de software 12.9

Na etapa de projeto em um sistema orientado a objetos, o projetista frequentemente determina que certas classes estão intimamente relacionadas. O projetista deve ‘fatorar’ atributos e comportamentos comuns e colocá-los em uma classe básica, depois utilizar a herança para formar classes derivadas, provendo-as de outras capacidades além das herdadas da classe básica.



Observação de engenharia de software 12.10

A criação de uma classe derivada não afeta o código-fonte da sua classe básica. A herança preserva a integridade de uma classe básica.



Observação de engenharia de software 12.11

Assim como os projetistas de sistemas não orientados a objetos devem evitar a proliferação de funções, os projetistas de sistemas orientados a objetos devem evitar proliferação de classes. A proliferação de classes cria problemas de gerenciamento e pode impedir a reusabilidade de software, porque se torna difícil para um cliente localizar a classe mais apropriada em uma enorme biblioteca de classes. A alternativa é criar menos classes que fornecem funcionalidades mais substanciais, mas essas classes talvez forneçam funcionalidades demais.



Dica de desempenho 12.3

Se as classes produzidas por herança forem maiores do que o necessário (isto é, contiverem funcionalidade demais), os recursos de memória e processamento podem ser desperdiçados. Herde da classe cujas funcionalidades estão ‘mais próximas’ daquilo que é necessário.

A leitura de definições de classes derivadas pode ser confusa, porque os membros herdados não são mostrados fisicamente nas classes derivadas, muito embora estejam presentes. Existe um problema semelhante ao documentar membros de classe derivada.

12.8 Síntese

Este capítulo introduziu a herança — a capacidade de criar uma classe absorvendo membros de dados e funções-membro de uma classe existente e aprimorando-os com novas capacidades. Por meio de uma série de exemplos que utilizam uma hierarquia de herança de empregados, você aprendeu as noções de classes básicas e classes derivadas e utilizou a herança `public` para criar uma classe derivada que herda membros de uma classe básica. O capítulo introduziu o especificador de acesso `protected`; as funções-membro de classe derivada podem acessar os membros `protected` da classe básica. Você aprendeu a acessar membros de classe básica redefinidos qualificando seus nomes com o nome da classe básica e o operador de resolução de escopo binário (`::`). Você também viu a ordem em que construtores e destrutores são chamados por objetos de classes que fazem parte de uma hierarquia de herança. Por fim, explicamos os três tipos de herança — `public`, `protected` e `private` — e a acessibilidade de membros de classe básica em uma classe derivada ao utilizar cada tipo.

No Capítulo 13, “Programação orientada a objetos: polimorfismo”, avançamos nossa discussão de herança introduzindo polimorfismo — um conceito orientado a objetos que permite escrever programas que tratam, de uma maneira mais geral, objetos de uma ampla variedade de classes relacionada por herança. Depois de estudar o Capítulo 13, você estará familiarizado com classes, objetos, encapsulamento, herança e polimorfismo — os aspectos essenciais da programação orientada a objetos.

Resumo

- A reutilização de software reduz o tempo e o custo de desenvolvimento de programas.
- A herança é uma forma de reutilização de software em que o programador cria uma classe que absorve dados e comportamentos de uma classe existente e os aprimora com novas capacidades. A classe existente é chamada de classe básica e a nova classe é referida como classe derivada.
- Uma classe básica direta é aquela a partir da qual uma classe derivada explicitamente herda (especificado pelo nome de classe à direita de `:` na primeira linha de uma definição de classe). Uma classe básica indireta é herdada de dois ou mais níveis acima na hierarquia de classes.
- Com a herança simples, uma classe é derivada de uma classe básica. Com a herança múltipla, uma classe herda de múltiplas classes básicas (possivelmente não relacionadas).
- Uma classe derivada representa um grupo mais especializado de objetos. Em geral, uma classe derivada contém comportamentos herdados de sua classe básica mais comportamentos adicionais. Uma classe derivada também pode personalizar comportamentos herdados da classe básica.
- Cada objeto de uma classe derivada é também um objeto da classe básica dessa classe. Entretanto, um objeto de classe básica não é um objeto dessa classe derivada.

- O relacionamento *é um* representa a herança. Em um relacionamento *é um*, um objeto de uma classe derivada também pode ser tratado como um objeto de sua classe básica.
- O relacionamento *tem um* representa a composição. Em um relacionamento *tem um*, um objeto contém um ou mais objetos de outras classes como membros, mas não expõe seu comportamento diretamente em sua interface.
- Uma classe derivada não pode diretamente acessar os membros `private` de sua classe básica; permitir isso violaria o encapsulamento da classe básica. Mas uma classe derivada pode acessar os membros `public` e `protected` de sua classe básica diretamente.
- Uma classe derivada pode produzir alterações de estado em membros da classe básica `private`, mas somente por funções-membro não-`private` fornecidas na classe básica e herdadas na classe derivada.
- Quando uma função-membro de classe básica é inadequada a uma classe derivada, ela pode ser redefinida na classe derivada com uma implementação apropriada.
- Os relacionamentos de herança simples formam estruturas hierárquicas do tipo árvore — existe uma classe básica em um relacionamento hierárquico com suas classes derivadas.
- É possível tratar objetos de classe básica e de classe derivada de modo semelhante; os aspectos comuns compartilhados entre os tipos de objeto são expressos nos membros de dados e funções-membro da classe básica.
- Os membros `public` de uma classe básica são acessíveis em qualquer lugar que o programa tiver um handle para um objeto dessa classe básica ou para um objeto de uma das classes derivadas dessa classe básica — ou, ao utilizar o operador de resolução de escopo binário, sempre que o nome da classe estiver no escopo.
- Os membros `private` de uma classe básica só são acessíveis dentro da definição dessa classe básica ou de amigas dessa classe.
- Os membros `protected` de uma classe básica têm um nível intermediário de proteção entre acesso `public` e `private`. Os membros `protected` de uma classe básica podem ser acessados por membros e amigos dessa classe básica e por membros e amigos de qualquer classe derivada dessa classe básica.
- Infelizmente, os membros de dados `protected` apresentam com frequência dois problemas importantes. Primeiro, o objeto de classe derivada não tem de utilizar uma função *set* para alterar o valor dos dados `protected` da classe básica. Segundo, é muito provável que as funções-membro da classe derivada dependam de detalhes de implementação da classe básica.
- Quando uma função-membro da classe derivada redefine uma função-membro da classe básica, a função-membro da classe básica pode ser acessada a partir da classe derivada qualificando o nome da função-membro da classe básica com o nome da classe básica e o operador de resolução de escopo binário (`::`).
- Quando um objeto de uma classe derivada é instanciado, o construtor da classe básica é chamado imediatamente (explícita ou implicitamente) para inicializar os membros de dados da classe básica no objeto da classe derivada (antes de os membros de dados da classe derivada serem inicializados).
- Declarar membros de dados `private`, e ao mesmo tempo fornecer funções-membro não-`private` para manipular e realizar a verificação de validação nesses dados, impõe boa engenharia de software.
- Quando um objeto de classe derivada é destruído, os destrutores são chamados na ordem inversa dos construtores — primeiro o destrutor de classe derivada e, depois, o destrutor de classe básica.
- Ao derivar uma classe de uma classe básica, a classe básica pode ser declarada como `public`, `protected` ou `private`.
- Ao derivar uma classe de uma classe `public` básica, os membros `public` da classe básica tornam-se membros `public` da classe derivada, e os membros `protected` da classe básica tornam-se membros `protected` da classe derivada.
- Ao derivar uma classe de uma classe `protected` básica, os membros `public` e `protected` da classe básica tornam-se os membros `protected` da classe derivada.
- Ao derivar uma classe de uma classe `private` básica, os membros `public` e `protected` da classe básica tornam-se os membros `private` da classe derivada.

Terminologia

classe básica	destrutor de classe derivada	inicializador de classe básica
classe básica direta	<i>é um</i> , relacionamento	nome qualificado
classe básica indireta	friend de uma classe básica	personalizar software
classe derivada	friend de uma classe derivada	<code>private</code> , classe básica
composição	herança	<code>private</code> , herança
construtor de classe básica	herança múltipla	<code>protected</code> , classe básica
construtor de classe derivada	herança única	<code>protected</code> , herança
construtor-padrão de classe básica	herdar os membros de uma classe existente	<code>protected</code> , membro, de uma classe
destrutor de classe básica	hierarquia de classes	<code>protected</code> , palavra-chave

public, classe básica	relacionamento hierárquico	subclasse
public, herança	software frágil	superclasse
redefinir uma função-membro de classe básica	software quebradiço	tem um, relacionamento

Exercícios de revisão

12.1 Preencha as lacunas em cada uma das seguintes sentenças:

- _____ é uma forma de reutilização de software em que novas classes absorvem os dados e comportamentos de classes existentes e aprimoram essas classes com novas capacidades.
- Os membros _____ de uma classe básica podem ser acessados somente na definição de classe básica ou nas definições de classe derivada.
- Em um relacionamento _____, um objeto de uma classe derivada também pode ser tratado como um objeto de sua classe básica.
- Em um relacionamento _____, um objeto de classe tem um ou mais objetos de outras classes como membros.
- Na herança simples, uma classe existe em um relacionamento _____ com suas classes derivadas.
- Os membros _____ de uma classe básica são acessíveis dentro dessa classe básica e em qualquer lugar que o programa tenha um handle para um objeto dessa classe básica ou para um objeto de uma de suas classes derivadas.
- Os membros de acesso `protected` de uma classe básica têm um nível de proteção entre aqueles de acesso `public` e _____.
- O C++ oferece _____, que permite a uma classe derivada herdar de muitas classes básicas, mesmo se essas classes básicas forem não relacionadas.
- Quando um objeto de uma classe derivada é instanciado, o _____ da classe básica é chamado implícita ou explicitamente para fazer qualquer inicialização necessária dos membros de dados de classe básica no objeto de classe derivada.
- Ao derivar uma classe de uma classe básica com a herança `public`, os membros `public` da classe básica tornam-se os membros _____ da classe derivada, e os membros `protected` da classe básica tornam-se os membros _____ da classe derivada.
- Ao derivar uma classe de uma classe básica com a herança `public`, os membros `protected` da classe básica tornam-se os membros _____ da classe derivada, e os membros `protected` da classe básica tornam-se os membros _____ da classe derivada.

12.2 Determine se cada uma das seguintes sentenças é verdadeira ou falsa. Se falsa, explique por quê.

- Os construtores de classe básica não são herdados por classes derivadas.
- Um relacionamento *tem um* é implementado via herança.
- Uma classe Carro tem um relacionamento *é um* com as classes Roda e Volante.
- A herança estimula a reutilização de software de alta qualidade comprovada.
- Quando um objeto de classe derivada é destruído, os destrutores são chamados na ordem inversa dos construtores.

Respostas dos exercícios de revisão

- 12.1** a) Herança. b) `protected`. c) *é um* ou herança. d) *tem um* ou composição ou agregação. e) hierárquico. f) `public`. g) `private`. h) herança múltipla. i) construtor. j) `public`, `protected`. k) `protected`, `protected`.
- 12.2** a) Verdadeira. b) Falsa. Um relacionamento *tem um* é implementado via composição. Um relacionamento *é um* é implementado via herança. c) Falsa. Esse é um exemplo de um relacionamento *tem um*. A classe Carro tem um relacionamento *é um* com a classe Veículo. d) Verdadeira. e) Verdadeira.

Exercícios

- 12.3** Muitos programas escritos com herança podem ser escritos com composição e vice-versa. Reescreva a classe `BasePlusCommissionEmployee` da hierarquia `CommissionEmployee–BasePlusCommissionEmployee` para utilizar composição em vez de herança. Depois de fazer isso, avalie os méritos relativos das duas abordagens para projetar as classes `CommissionEmployee` e `BasePlusCommissionEmployee`, bem como para programas orientados a objetos em geral. Que abordagem é mais natural? Por quê?
- 12.4** Discuta de que maneira a herança promove a reutilização de software, economiza tempo durante o desenvolvimento de programa e ajuda a evitar erros.
- 12.5** Alguns programadores preferem não utilizar o acesso `protected` porque acreditam que ele quebra o encapsulamento da classe básica. Discuta os méritos relativos a usar acesso `protected` versus acesso `private` em classes básicas.
- 12.6** Desenhe uma hierarquia de herança para alunos universitários semelhante à hierarquia mostrada na Figura 12.2. Utilize `Aluno` como a classe básica da hierarquia, então inclua as classes `AlunoDeGraduação` e `AlunoGraduado` que derivam de `Aluno`. Continue a estender a hierarquia o mais profundamente (isto é, com muitos níveis) possível. Por exemplo, `Primeiranistas`, `Segundanistas`, `Terceiranistas` e `Quartanistas` poderiam derivar de `AlunoDeGraduação`; e `AlunoDeDoutorado` e `AlunoDeMestrado` poderiam derivar de `AlunoGraduado`. Depois de desenhar a hierarquia, discuta os relacionamentos entre as classes. [Nota: Você não precisa escrever nenhum código para este exercício.]

- 12.7** O mundo das formas é muito mais rico que as formas incluídas na hierarquia de herança da Figura 12.3. Anote todas as formas que puder imaginar — bidimensionais e tridimensionais — e as forme em uma hierarquia Shape mais completa com o maior número de níveis possível. Sua hierarquia deve ter a classe básica Forma a partir da qual a classe FormaBiDimensional e a FormaTriDimensional são derivadas. [Nota: Você não precisa escrever nenhum código para este exercício.] Utilizaremos essa hierarquia nos exercícios do Capítulo 13 para processar um conjunto de formas distintas como objetos da classe básica Forma. (Essa técnica, chamada polimorfismo, é o assunto do Capítulo 13.)
- 12.8** Desenhe uma hierarquia de herança para as classes Quadrilátero, Trapezóide, Paralelograma, Retângulo e Quadrado. Utilize Quadrilátero como a classe básica da hierarquia. Torne a hierarquia o mais profunda possível.
- 12.9** (*Hierarquia de herança de Package*) Os serviços de correio expresso, como FedEx®, DHL® e UPS®, oferecem várias opções de entrega, cada qual com custos específicos. Crie uma hierarquia de herança para representar vários tipos de pacotes. Utilize Package como a classe básica da hierarquia, então inclua as classes TwoDayPackage e OvernightPackage que derivam de Package. A classe básica Package deve incluir membros de dados que representam nome, endereço, cidade, estado e CEP tanto do remetente como do destinatário do pacote, além dos membros de dados que armazenam o peso (em quilos) e o custo por quilo para a entrega do pacote. O construtor Package deve inicializar esses membros de dados. Assegure que o peso e o custo por quilo conttenham valores positivos. Package deve fornecer uma função-membro public calculateCost que retorna um double indicando o custo associado com a entrega do pacote. A função calculateCost de Package deve determinar o custo multiplicando o peso pelo custo (em quilos). A classe derivada TwoDayPackage deve herdar a funcionalidade da classe básica Package, mas também incluir um membro de dados que representa uma taxa fixa que a empresa de entrega cobra pelo serviço de entrega de dois dias. O construtor TwoDayPackage deve receber um valor para inicializar esse membro de dados. TwoDayPackage deve redefinir a função-membro calculateCost para que ela calcule o custo de entrega adicionando a taxa fixa ao custo baseado em peso calculado pela função calculateCost da classe básica Package. A classe OvernightPackage deve herdar diretamente da classe Package e conter um membro de dados adicional para representar uma taxa adicional por quilo cobrado pelo serviço de entrega noturno. OvernightPackage deve redefinir a função-membro calculateCost para que ela acrescente a taxa adicional por quilo ao custo-padrão por quilo antes de calcular o custo da entrega. Escreva um programa de teste que cria objetos de todos os tipos de Package e testa a função-membro calculateCost.
- 12.10** (*Hierarquia de herança de Account*) Crie uma hierarquia de herança que um banco possa utilizar para representar as contas bancárias dos clientes. Todos os clientes nesse banco podem depositar (isto é, creditar) dinheiro em suas contas e retirar (isto é, debitar) o dinheiro delas. Há também tipos mais específicos de contas. As contas de poupança, por exemplo, recebem juros pelo dinheiro depositado nelas. As contas bancárias, por outro lado, cobram uma taxa por transação (isto é, crédito ou débito).
- Crie uma hierarquia de herança contendo classe básica Account e classes derivadas SavingsAccount e CheckingAccount que herdam da classe Account. A classe básica Account deve incluir um membro de dados do tipo double para representar o saldo da conta. A classe deve fornecer um construtor que recebe um saldo inicial e o utiliza para inicializar o membro de dados. O construtor deve validar o saldo inicial para assegurar que ele é maior que ou igual a 0.0. Caso contrário, o saldo deve ser configurado como 0.0 e o construtor deve exibir uma mensagem de erro, indicando que o saldo inicial era inválido. A classe deve fornecer três funções-membro. A função-membro credit deve adicionar uma quantia ao saldo atual. A função-membro debit deve retirar dinheiro de Account e assegurar que o valor do débito não exceda o saldo de Account. Se exceder, o saldo deve permanecer inalterado e a função deve imprimir a mensagem "Debit amount exceeded account balance" [Saldo insuficiente]. A função-membro getBalance deve retornar o saldo atual.
- A classe derivada SavingsAccount deve herdar a funcionalidade de uma Account, mas também incluir um membro de dados do tipo double para indicar a taxa de juros (porcentagem) atribuída à Account. O construtor SavingsAccount deve receber o saldo inicial, bem como um valor inicial para a taxa de juros de SavingsAccount. SavingsAccount deve fornecer uma função-membro public calculateInterest que retorna um double para indicar os juros auferidos por uma conta. A função-membro calculateInterest deve determinar esse valor multiplicando a taxa de juros pelo saldo da conta. [Nota: SavingsAccount deve herdar as funções-membro credit e debit exatamente como são sem redefini-las.]
- A classe derivada CheckingAccount deve herdar da classe básica Account e incluir um membro adicional de dados do tipo double que representa a taxa cobrada por transação. O construtor CheckingAccount deve receber o saldo inicial, bem como um parâmetro que indica o valor de uma taxa. A classe CheckingAccount deve redefinir as funções-membro credit e debit para que subtraíam a taxa do saldo da conta sempre que qualquer uma das transações for realizada com sucesso. As versões CheckingAccount dessas funções devem invocar a versão Account da classe básica para realizar as atualizações de saldo de uma conta. A função debit CheckingAccount deve cobrar uma taxa somente se o dinheiro for realmente retirado (isto é, o valor do débito não exceder ao do saldo da conta). [Dica: Defina a função debit de Account para que ela retorne um bool indicando se houve retirada de dinheiro. Em seguida, utilize o valor de retorno para determinar se uma taxa deve ser cobrada.]
- Depois de definir as classes nessa hierarquia, escreva um programa que cria objetos de cada classe e testa suas funções-membro. Adicione os juros ao objeto SavingsAccount invocando primeiro sua função calculateInterest e, então, passando o valor retornado dos juros para a função credit do objeto.



*Um Anel para todos governar,
Um Anel para encontrá-los,
Um Anel para todos reunir e na
escuridão aprisioná-los.*
John Ronald Reuel Tolkien

*O silêncio muitas vezes de pura
inocência
Convence quando a fala falha.*
William Shakespeare

*Propostas genéricas não
decidem casos concretos.*
Oliver Wendell Holmes

*Um filósofo de estatura
imponente não pensa em um
vazio. Mesmo suas idéias mais
abstratas são, em alguma
medida, condicionadas pelo que
é ou não conhecido na época
em que ele vive.*
Alfred North Whitehead

Programação orientada a objetos: polimorfismo

OBJETIVOS

Neste capítulo, você aprenderá:

- O que é polimorfismo, como ele torna a programação mais conveniente e os sistemas mais extensíveis e sustentáveis.
- A declarar e utilizar funções `virtual` para produzir polimorfismo.
- A distinguir entre classes abstratas e classes concretas.
- A declarar funções `virtual` puras para criar classes abstratas.
- A utilizar informações de tipo em tempo de execução (*Run-Time Type Information* – RTTI) com `downcasting`, `dynamic_cast`, `typeid` e `type_info`.
- Como o C++ implementa as funções `virtual` e a vinculação dinâmica 'sob o capô'.
- A utilizar os destrutores `virtual` para assegurar que todos os destrutores apropriados executem em um objeto.

Sumário

13.1

Introdução

13.2

Exemplos de polimorfismo

13.3

Relacionamentos entre objetos em uma hierarquia de herança

13.3.1

Invocando funções de classe básica a partir de objetos de classe derivada

13.3.2

Apontando ponteiros de classe derivada para objetos da classe básica

13.3.3

Chamadas de função-membro de classe derivada via ponteiros de classe básica

13.3.4

Funções virtuais

13.3.5

Resumo das atribuições permitidas entre objetos de classe básica e de classe derivada e ponteiros

13.4

Campos de tipo e instruções switch

13.5

Classes abstratas e funções virtual puras

13.6

Estudo de caso: sistema de folha de pagamento utilizando polimorfismo

13.6.1

Criando a classe básica abstrata Employee

13.6.2

Criando a classe derivada concreta SalariedEmployee

13.6.3

Criando a classe derivada concreta HourlyEmployee

13.6.4

Criando a classe derivada concreta CommissionEmployee

13.6.5

Criando a classe derivada concreta indireta BasePlusCommissionEmployee

13.6.6

Demonstrando o processamento polimórfico

13.7

Polimorfismo, funções virtual e vinculação dinâmica ‘sob o capô’ (opcional)

13.8

Estudo de caso: sistema de folha de pagamento utilizando polimorfismo e informações de tipo em tempo de execução com downcasting, dynamic_cast, typeid e type_info

13.9

Destrutores virtuais

13.10

Estudo de caso de engenharia de software: incorporando herança ao sistema ATM (opcional)

13.11

Síntese

Resumo

Terminologia

Exercícios de revisão

Respostas dos exercícios de revisão

Exercícios

13.1 Introdução

Nos capítulos 9–12, discutimos as tecnologias-chave de programação orientada a objetos incluindo classes, objetos, encapsulamento, sobrecarga de operadores e herança. Agora continuamos nosso estudo de OOP explicando e demonstrando o **polimorfismo** com hierarquias de herança. O polimorfismo permite ‘programar no geral’ em vez de ‘programar no específico’. Em particular, o polimorfismo permite escrever programas que processam objetos de classes que fazem parte da mesma hierarquia de classes como se todos fossem objetos da classe básica da hierarquia. Como veremos em breve, o polimorfismo se livra dos handles de ponteiro de classe básica e dos handles de referência de classe básica, mas não dos handles de nome.

Considere o exemplo de polimorfismo a seguir. Suponha que criamos um programa que simula o movimento de vários tipos de animais para um estudo biológico. As classes Peixe, Anfíbio e Pássaro representam os três tipos de animais sob investigação. Imagine que cada uma dessas classes herda da classe básica Animal, que contém uma função mover e mantém a localização atual de um animal. Toda classe derivada implementa a função mover. Nosso programa mantém um vetor de ponteiros para objetos das várias classes derivadas de Animal. Para simular os movimentos dos animais, o programa envia a mesma mensagem a cada objeto uma vez por segundo — a saber, mover. Entretanto, cada tipo específico de Animal responde a uma mensagem mover de maneira própria e única — um Peixe poderia nadar dois metros, um Sapo poderia pular três metros e um Pássaro, voar dez metros. O programa emite a mesma mensagem (isto é, mover) para cada objeto animal genericamente, mas cada objeto sabe como modificar sua posição apropriadamente de acordo com seu tipo de movimento específico. Contar com o fato de que cada objeto sabe ‘fazer a coisa certa’ (isto é, faz o que é apropriado a esse tipo de objeto) em resposta à mesma chamada de método é o conceito-chave do polimorfismo. A mesma mensagem (nesse caso, mover) enviada a uma variedade de objetos tem ‘muitas formas’ de resultados — daí o termo polimorfismo.

Com o polimorfismo podemos projetar e implementar sistemas que são facilmente extensíveis — novas classes podem ser adicionadas com pouca ou nenhuma modificação a partes gerais do programa, contanto que as novas classes façam parte da hierarquia de herança que o programa processa genericamente. As únicas partes de um programa que devem ser alteradas para acomodar as novas classes são aquelas que exigem conhecimento direto das novas classes que o programador adiciona à hierarquia. Por exemplo, se criarmos a classe Tartaruga que herda da classe Animal (que poderia responder a uma mensagem mover rastejando cinco centímetros), precisaremos escrever somente a classe Tartaruga e a parte da simulação que instancia um objeto Tartaruga. As partes da simulação que processam cada Animal genericamente podem permanecer idênticas.

Começamos com uma sequência de pequenos exemplos focalizados que levam a um entendimento das funções `virtual` e da vinculação dinâmica — duas tecnologias subjacentes do polimorfismo. Então apresentamos um estudo de caso que revê a hierarquia `Employee` do Capítulo 12. No estudo de caso, definimos uma ‘interface’ comum (isto é, um conjunto de funcionalidades) para todas as classes na hierarquia. Essas funcionalidades comuns entre empregados são definidas na chamada classe básica abstrata, `Employee`, da qual as classes `SalariedEmployee`, `HourlyEmployee` e `CommissionEmployee` herdam diretamente, e a classe `BaseCommissionEmployee` herda indiretamente. Logo veremos o que torna uma classe ‘abstrata’ ou seu oposto — ‘concreta’.

Nessa hierarquia, cada empregado tem uma função `earnings` para calcular o salário semanal do empregado. Essas funções `earnings` variam por tipo de empregado — por exemplo, `SalariedEmployees` recebem um salário semanal fixo independentemente do número de horas trabalhadas, enquanto `HourlyEmployees` são pagos por hora e recebem pagamento de horas extras. Mostramos como processar cada empregado ‘de maneira mais geral’ — isto é, utilizando ponteiros de classe básica para chamar a função `earnings` de vários objetos de classe derivada. Desse modo, o programador precisa se preocupar apenas com um tipo de chamada de função, que pode ser utilizada para executar várias funções diferentes com base nos objetos referenciados pelos ponteiros de classe básica.

Um recurso-chave deste capítulo é sua discussão detalhada (opcional) de polimorfismo, funções `virtual` e vinculação dinâmica ‘sob o capô’, que utiliza um diagrama detalhado para explicar como o polimorfismo pode ser implementado no C++.

Ocasionalmente, ao realizar processamento polimórfico, precisamos programar ‘no específico’, o que significa que precisam ser realizadas operações em um tipo de objeto específico em uma hierarquia — a operação não pode ser aplicada de maneira geral aos vários tipos de objetos. Reutilizamos nossa hierarquia `Employee` para demonstrar as poderosas capacidades das **informações de tipo em tempo de execução** (*run-time type information – RTTI*) e a **coerção dinâmica**, que permite a um programa determinar o tipo de objeto em tempo de execução e agir sobre esse objeto de maneira correspondente. Utilizamos essas capacidades para determinar se um objeto empregado particular é um `BasePlusCommissionEmployee`, então damos para esse empregado um bônus de 10% em seu salário-base.

13.2 Exemplos de polimorfismo

Nesta seção, discutimos vários exemplos de polimorfismo. Com polimorfismo, uma função pode produzir a ocorrência de ações diferentes, dependendo do tipo do objeto em que a função é invocada. Isso fornece tremenda capacidade expressiva para o programador. Se a classe `Retângulo` é derivada da classe `Quadrilátero`, então um objeto `Retângulo` é uma versão mais específica de um objeto `Quadrilátero`. Portanto, qualquer operação (como calcular o perímetro ou a área) que pode ser realizada em um objeto da classe `Quadrilátero` também pode ser realizada em um objeto da classe `Retângulo`. Tais operações também podem ser realizadas em outros tipos de `Quadriláteros`, como `Quadrados`, `Paralelogramas` e `Trapezóides`. O polimorfismo ocorre quando um programa invoca uma função `virtual` por meio de um ponteiro de classe básica (isto é, `Quadrilátero`) ou referência — o C++ escolhe dinamicamente (isto é, em tempo de execução) a função correta para a classe a partir da qual o objeto foi instanciado. Veremos um exemplo de código que ilustra esse processo na Seção 13.3.

Como outro exemplo, suponha que vamos projetar um videogame que manipula objetos de vários tipos diferentes, incluindo objetos das classes `Marciano`, `Venusiano`, `Plutoniano`, `NaveEspacial` e `CanhaoDeLaser`. Imagine que cada uma dessas classes herda da classe básica comum `ObjetoEspacial`, que contém a função-membro `desenhar`. Toda classe derivada implementa essa função de maneira apropriada a essa classe. Um programa gerenciador de tela mantém um contêiner (por exemplo, um `vector`) que armazena ponteiros de `ObjetoEspacial` para objetos das várias classes. Para atualizar a tela, o gerenciador de tela envia periodicamente a mesma mensagem a cada objeto — a saber, `desenhar`. Cada tipo de objeto responde de maneira única. Por exemplo, um objeto `Marciano` desenharia a si mesmo em vermelho com o número apropriado de antenas. Um objeto `NaveEspacial` desenharia a si mesmo como um disco voador brilhante. Um objeto `CanhaoDeLaser` poderia se desenhar como um feixe vermelho brilhante através da tela. Mais uma vez, a mesma mensagem (nesse caso, `desenhar`) enviada a uma variedade de objetos tem ‘muitas formas’ de resultados.

Um gerenciador de tela polimórfico facilita adicionar novas classes a um sistema com modificações mínimas no seu código. Suponha que quiséssemos adicionar objetos da classe `Mercuriano` ao nosso videogame. Para fazer isso, devemos construir uma classe `Mercuriano` que herda de `ObjetoEspacial`, mas fornece sua própria definição de função-membro `desenhar`. Então, quando ponteiros para objetos da classe `Mercuriano` aparecem no contêiner, o programador não precisa modificar o código para o gerenciador de tela. O gerenciador de tela invoca a função-membro `desenhar` em cada objeto no contêiner, independentemente do tipo de objeto, então os novos objetos `Mercurianos` simplesmente se ‘conectam’. Portanto, sem modificar o sistema (além de construir e incluir as próprias classes), os programadores podem utilizar o polimorfismo para acomodar classes adicionais, inclusive aquelas que nem mesmo foram consideradas quando o sistema foi criado.



Observação de engenharia de software 13.1

Com as funções `virtual` e o polimorfismo, você pode tratar generalidades e deixar a questão do ambiente de tempo de execução em si para as especificidades. Você pode instruir uma variedade de objetos a se comportar de maneiras apropriadas a esses objetos sem mesmo conhecer seus tipos (contanto que esses objetos pertençam à mesma hierarquia de herança e estejam sendo acessados a partir de um ponteiro de classe básica comum).



Observação de engenharia de software 13.2

O polimorfismo promove extensibilidade: o software escrito para invocar comportamento polimórfico é escrito independentemente dos tipos dos objetos para os quais as mensagens são enviadas. Portanto, novos tipos de objetos que podem responder a mensagens existentes podem ser incorporados nesse sistema sem modificar o sistema de base. Somente o código de cliente que instancia os novos objetos deve ser modificado para acomodar os novos tipos.

13.3 Relacionamentos entre objetos em uma hierarquia de herança

A Seção 12.4 criou uma hierarquia de classes de empregados, em que a classe `BasePlusCommissionEmployee` herdou da classe `CommissionEmployee`. Os exemplos do Capítulo 12 manipularam os objetos `CommissionEmployee` e `BasePlusCommissionEmployee` utilizando os nomes dos objetos para invocar suas funções-membro. Agora examinamos os relacionamentos entre classes em uma hierarquia mais rigorosamente. As próximas várias seções apresentam uma série de exemplos que demonstram como ponteiros das classes básicas e derivadas podem ser apontados para objetos das classes básicas e derivadas e como esses ponteiros podem ser utilizados para invocar funções-membro que manipulam esses objetos. Perto do final desta seção, demonstramos como obter o comportamento polimórfico a partir de ponteiros de classes básicas apontados para objetos de classes derivadas.

Na Seção 13.3.1 atribuímos o endereço de um objeto de classe derivada a um ponteiro de classe básica, então mostramos que invocar uma função via ponteiro de classe básica invoca as funcionalidades da classe básica — isto é, o tipo do handle determina que função é chamada. Na Seção 13.3.2, atribuímos o endereço de um objeto de classe básica a um ponteiro de classe derivada, o que resulta em um erro de compilação. Discutimos a mensagem de erro e investigamos por que o compilador não permite essa atribuição. Na Seção 13.3.3, atribuímos o endereço de um objeto de classe derivada a um ponteiro de classe básica, depois examinamos como o ponteiro de classe básica pode ser utilizado para invocar apenas as funcionalidades de classe básica — quando tentamos invocar funções-membro de classe derivada pelo ponteiro de classe básica, ocorrem erros de compilação. Por fim, na Seção 13.3.4, introduzimos as funções `virtual` e o polimorfismo declarando uma função de classe básica como `virtual`. Então atribuímos um objeto de classe derivada ao ponteiro de classe básica e utilizamos esse ponteiro para invocar as funcionalidades da classe derivada — exatamente a capacidade de que precisamos para alcançar o comportamento polimórfico.

Um conceito-chave nesses exemplos é demonstrar que um objeto de uma classe derivada pode ser tratado como um objeto de sua classe básica. Isso permite várias manipulações interessantes. Por exemplo, um programa pode criar um array de ponteiros de classe básica que aponta para objetos de muitos tipos de classes derivadas. Apesar de os objetos de classes derivadas serem de tipos diferentes, o compilador permite isso porque cada objeto de classe derivada *é um* objeto de sua classe básica. Entretanto, não podemos tratar um objeto de classe básica como um objeto de qualquer uma de suas classes derivadas. Por exemplo, um `CommissionEmployee` não é um `BasePlusCommissionEmployee` na hierarquia definida no Capítulo 12 — um `CommissionEmployee` não tem um membro de dados `baseSalary` e não tem as funções-membro `setBaseSalary` e `getBaseSalary`. O relacionamento *é um* se aplica somente de uma classe derivada para suas classes básicas diretas e indiretas.

13.3.1 Invocando funções de classe básica a partir de objetos de classe derivada

O exemplo nas figuras 13.1–13.5 demonstra três maneiras de apontar ponteiros de classe básica e ponteiros de classe derivada em objetos de classe básica e de classe derivada. As duas primeiras são simples e diretas — apontamos um ponteiro de classe básica para um objeto de classe básica (e invocamos as funcionalidades da classe básica) e apontamos um ponteiro de classe derivada para um objeto de classe derivada (e invocamos as funcionalidades da classe derivada). Em seguida, demonstramos o relacionamento entre as classes derivadas e básicas (isto é, o relacionamento de herança *é um*) apontando um ponteiro de classe básica para um objeto de classe derivada (e mostrando que as funcionalidades de classe básica estão, de fato, disponíveis no objeto de classe derivada).

A classe `CommissionEmployee` (figuras 13.1–13.2), que discutimos no Capítulo 12, é utilizada para representar os empregados que recebem uma porcentagem de suas vendas. A classe `BasePlusCommissionEmployee` (figuras 13.3–13.4), que também discutimos no Capítulo 12, é utilizada para representar os empregados que recebem um salário-base mais uma porcentagem de suas vendas. Todo objeto `BasePlusCommissionEmployee` *é um* `CommissionEmployee` que também tem um salário-base. A função-membro `earnings` da classe `BasePlusCommissionEmployee` (linhas 32–35 da Figura 13.4) redefine a função-membro `earnings` da classe `CommissionEmployee` (linhas 79–82 da Figura 13.2) para incluir o salário-base do objeto. A função-membro `print` da classe `BasePlusCommissionEmployee` (linhas 38–46 da Figura 13.4) redefine a função-membro `print` da classe `CommissionEmployee` (linhas 85–92 da Figura 13.2) para exibir as mesmas informações que a função `print` exibe na classe `CommissionEmployee`, bem como o salário-base do empregado.

Na Figura 13.5, as linhas 19–20 criam um objeto `CommissionEmployee` e a linha 23 cria um ponteiro para um objeto `CommissionEmployee`; as linhas 26–27 criam um objeto `BasePlusCommissionEmployee` e a linha 30 cria um ponteiro para um objeto `BasePlusCommissionEmployee`. As linhas 37 e 39 utilizam o nome de cada objeto (`commissionEmployee` e `basePlusCommissionEmployee`, respectivamente) para invocar a função-membro `print` de cada objeto. A linha 42 atribui o endereço do objeto da classe básica `commissionEmployee` ao ponteiro de classe básica `commissionEmployeePtr`, que a linha 45 utiliza para invocar a função-membro `print` nesse objeto `CommissionEmployee`. Isso invoca a versão de `print` definida na classe básica `CommissionEmployee`. De modo semelhante, a linha 48 atribui o endereço do objeto da classe derivada `basePlusCommissionEmployee` ao ponteiro de classe derivada `basePlusCommissionEmployeePtr`, que a linha 52 utiliza para invocar a função-membro `print` nesse objeto `BasePlusCommissionEmployee`. Isso invoca a versão de `print` definida na classe derivada `BasePlusCommissionEmployee`. A linha 55 então atribui o endereço de objeto

```

1 // Figura 13.1: CommissionEmployee.h
2 // Classe CommissionEmployee representa um empregado comissionado.
3 #ifndef COMMISSION_H
4 #define COMMISSION_H
5
6 #include <string> // classe string padrão C++
7 using std::string;
8
9 class CommissionEmployee
10 {
11 public:
12     CommissionEmployee( const string &, const string &, const string &,
13                        double = 0.0, double = 0.0 );
14
15     void setFirstName( const string & ); // configura o nome
16     string getFirstName() const; // retorna o nome
17
18     void setLastName( const string & ); // configura o sobrenome
19     string getLastName() const; // retorna o sobrenome
20
21     void setSocialSecurityNumber( const string & ); // configura o SSN
22     string getSocialSecurityNumber() const; // retorna o SSN
23
24     void setGrossSales( double ); // configura a quantidade de vendas brutas
25     double getGrossSales() const; // retorna a quantidade de vendas brutas
26
27     void setCommissionRate( double ); // configura a taxa de comissão
28     double getCommissionRate() const; // retorna a taxa de comissão
29
30     double earnings() const; // calcula os rendimentos
31     void print() const; // imprime o objeto CommissionEmployee
32 private:
33     string firstName;
34     string lastName;
35     string socialSecurityNumber;
36     double grossSales; // vendas brutas semanais
37     double commissionRate; // porcentagem da comissão
38 }; // fim da classe CommissionEmployee
39
40 #endif

```

Figura 13.1 Arquivo de cabeçalho da classe CommissionEmployee.

```

1 // Figura 13.2: CommissionEmployee.cpp
2 // Definições de função-membro da classe CommissionEmployee.
3 #include <iostream>
4 using std::cout;
5
6 #include "CommissionEmployee.h" // definição da classe CommissionEmployee
7
8 // construtor
9 CommissionEmployee::CommissionEmployee(
10     const string &first, const string &last, const string &ssn,
11     double sales, double rate )

```

Figura 13.2 Arquivo de implementação da classe CommissionEmployee.

(continua)

```

12     : firstName( first ), lastName( last ), socialSecurityNumber( ssn )
13 {
14     setGrossSales( sales ); // valida e armazena as vendas brutas
15     setCommissionRate( rate ); // valida e armazena a taxa de comissão
16 } // fim do construtor CommissionEmployee
17
18 // configura o nome
19 void CommissionEmployee::setFirstName( const string &first )
20 {
21     firstName = first; // deve validar
22 } // fim da função setFirstName
23
24 // retorna o nome
25 string CommissionEmployee::getFirstName() const
26 {
27     return firstName;
28 } // fim da função getFirstName
29
30 // configura o sobrenome
31 void CommissionEmployee::setLastName( const string &last )
32 {
33     lastName = last; // deve validar
34 } // fim da função setLastName
35
36 // retorna o sobrenome
37 string CommissionEmployee::getLastName() const
38 {
39     return lastName;
40 } // fim da função getLastName
41
42 // configura o SSN
43 void CommissionEmployee::setSocialSecurityNumber( const string &ssn )
44 {
45     socialSecurityNumber = ssn; // deve validar
46 } // fim da função setSocialSecurityNumber
47
48 // retorna o SSN
49 string CommissionEmployee::getSocialSecurityNumber() const
50 {
51     return socialSecurityNumber;
52 } // fim da função getSocialSecurityNumber
53
54 // configura a quantidade de vendas brutas
55 void CommissionEmployee::setGrossSales( double sales )
56 {
57     grossSales = ( sales < 0.0 ) ? 0.0 : sales;
58 } // fim da função setGrossSales
59
60 // retorna a quantidade de vendas brutas
61 double CommissionEmployee::getGrossSales() const
62 {
63     return grossSales;
64 } // fim da função getGrossSales
65
66 // configura a taxa de comissão
67 void CommissionEmployee::setCommissionRate( double rate )
68 {

```

Figura 13.2 Arquivo de implementação da classe CommissionEmployee.

(continua)

```

69     commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
70 } // fim da função setCommissionRate
71
72 // retorna a taxa de comissão
73 double CommissionEmployee::getCommissionRate() const
74 {
75     return commissionRate;
76 } // fim da função getCommissionRate
77
78 // calcula os rendimentos
79 double CommissionEmployee::earnings() const
80 {
81     return getCommissionRate() * getGrossSales();
82 } // fim da função earnings
83
84 // imprime o objeto CommissionEmployee
85 void CommissionEmployee::print() const
86 {
87     cout << "commission employee: "
88         << getFirstName() << ' ' << getLastName()
89         << "\nsocial security number: " << getSocialSecurityNumber()
90         << "\ngross sales: " << getGrossSales()
91         << "\ncommission rate: " << getCommissionRate();
92 } // fim da função print

```

Figura 13.2 Arquivo de implementação da classe CommissionEmployee.

(continuação)

```

1 // Figura 13.3: BasePlusCommissionEmployee.h
2 // Classe BasePlusCommissionEmployee derivada da classe
3 // CommissionEmployee.
4 #ifndef BASEPLUS_H
5 #define BASEPLUS_H
6
7 #include <string> // classe string padrão C++
8 using std::string;
9
10 #include "CommissionEmployee.h" // declaração da classe CommissionEmployee
11
12 class BasePlusCommissionEmployee : public CommissionEmployee
13 {
14 public:
15     BasePlusCommissionEmployee( const string &, const string &,
16                               const string &, double = 0.0, double = 0.0, double = 0.0 );
17
18     void setBaseSalary( double ); // configura o salário-base
19     double getBaseSalary() const; // retorna o salário-base
20
21     double earnings() const; // calcula os rendimentos
22     void print() const; // imprime o objeto BasePlusCommissionEmployee
23 private:
24     double baseSalary; // salário-base
25 }; // fim da classe BasePlusCommissionEmployee
26
27 #endif

```

Figura 13.3 Arquivo de cabeçalho da classe BasePlusCommissionEmployee.

```

1 // Figura 13.4: BasePlusCommissionEmployee.cpp
2 // Definições de função-membro da classe BasePlusCommissionEmployee.
3 #include <iostream>
4 using std::cout;
5
6 // Definição da classe BasePlusCommissionEmployee
7 #include "BasePlusCommissionEmployee.h"
8
9 // construtor
10 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
11     const string &first, const string &last, const string &ssn,
12     double sales, double rate, double salary )
13     // chama explicitamente o construtor da classe básica
14     : CommissionEmployee( first, last, ssn, sales, rate )
15 {
16     setBaseSalary( salary ); // valida e armazena o salário-base
17 } // fim do construtor BasePlusCommissionEmployee
18
19 // configura o salário-base
20 void BasePlusCommissionEmployee::setBaseSalary( double salary )
21 {
22     baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
23 } // fim da função setBaseSalary
24
25 // retorna o salário-base
26 double BasePlusCommissionEmployee::getBaseSalary() const
27 {
28     return baseSalary;
29 } // fim da função getBaseSalary
30
31 // calcula os rendimentos
32 double BasePlusCommissionEmployee::earnings() const
33 {
34     return getBaseSalary() + CommissionEmployee::earnings();
35 } // fim da função earnings
36
37 // imprime o objeto BasePlusCommissionEmployee
38 void BasePlusCommissionEmployee::print() const
39 {
40     cout << "base-salaried ";
41
42     // invoca a função print de CommissionEmployee
43     CommissionEmployee::print();
44
45     cout << "\nbase salary: " << getBaseSalary();
46 } // fim da função print

```

Figura 13.4 Arquivo de implementação da classe BasePlusCommissionEmployee.

de classe derivada `basePlusCommissionEmployee` ao ponteiro de classe básica `commissionEmployeePtr`, que a linha 59 utiliza para invocar a função-membro `print`. O compilador C++ permite esse ‘cruzamento’ (*crossover*) porque um objeto de uma classe derivada é um objeto de sua classe básica. Observe que, apesar de o ponteiro de classe básica `CommissionEmployee` apontar para um objeto `BasePlusCommissionEmployee` de classe derivada, a função-membro `print` da classe básica `CommissionEmployee` é invocada (em vez da função `print` de `BasePlusCommissionEmployee`). A saída de cada invocação de função-membro `print` nesse programa revela que *as funcionalidades invocadas dependem do tipo do handle (isto é, o tipo de ponteiro ou referência) utilizado para invocar a função, não do tipo do objeto para o qual o handle aponta*. Na Seção 13.3.4, quando introduzimos as funções `virtual`, demonstramos que é possível invocar as funcionalidades do tipo de objeto, em vez de invocar as funcionalidades do tipo de handle. Veremos que isso é crucial para implementar o comportamento polimórfico — o tópico-chave deste capítulo.

```

1 // Figura 13.5: fig13_05.cpp
2 // Apontando ponteiros de classe básica e classe derivada para objetos de classe
3 // básica e classe derivada, respectivamente.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7 using std::fixed;
8
9 #include <iomanip>
10 using std::setprecision;
11
12 // inclui definições de classe
13 #include "CommissionEmployee.h"
14 #include "BasePlusCommissionEmployee.h"
15
16 int main()
17 {
18     // cria objeto de classe básica
19     CommissionEmployee commissionEmployee(
20         "Sue", "Jones", "222-22-2222", 10000, .06 );
21
22     // cria ponteiro de classe básica
23     CommissionEmployee *commissionEmployeePtr = 0;
24
25     // cria objeto de classe derivada
26     BasePlusCommissionEmployee basePlusCommissionEmployee(
27         "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
28
29     // cria ponteiro de classe derivada
30     BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;
31
32     // configura a formatação de saída de ponto flutuante
33     cout << fixed << setprecision( 2 );
34
35     // gera saída dos objetos commissionEmployee e basePlusCommissionEmployee
36     cout << "Print base-class and derived-class objects:\n\n";
37     commissionEmployee.print(); // invoca print da classe básica
38     cout << "\n\n";
39     basePlusCommissionEmployee.print(); // invoca print da classe derivada
40
41     // aponta o ponteiro de classe básica para o objeto de classe básica e imprime
42     commissionEmployeePtr = &commissionEmployee; // perfeitamente natural
43     cout << "\n\nCalling print with base-class pointer to "
44         << "\nbase-class object invokes base-class print function:\n\n";
45     commissionEmployeePtr->print(); // invoca print da classe básica
46
47     // aponta o ponteiro de classe derivada para o objeto de classe derivada e imprime
48     basePlusCommissionEmployeePtr = &basePlusCommissionEmployee; // natural
49     cout << "\n\nCalling print with derived-class pointer to "
50         << "\nderived-class object invokes derived-class "
51         << "print function:\n\n";
52     basePlusCommissionEmployeePtr->print(); // invoca print da classe derivada
53
54     // aponta ponteiro de classe básica para o objeto de classe derivada e imprime
55     commissionEmployeePtr = &basePlusCommissionEmployee;
56     cout << "\n\nCalling print with base-class pointer to "
57         << "derived-class object\ninvokes base-class print "

```

Figura 13.5 Atribuindo endereços de objetos das classes básica e derivada aos ponteiros de classe básica e derivada.

(continua)

```

58     << "function on that derived-class object:\n\n";
59     commissionEmployeePtr->print(); // invoca print da classe básica
60     cout << endl;
61     return 0;
62 } // fim de main

```

Print base-class and derived-class objects:

```

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

```

```

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00

```

Calling print with base-class pointer to
base-class object invokes base-class print function:

```

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

```

Calling print with derived-class pointer to
derived-class object invokes derived-class print function:

```

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00

```

Calling print with base-class pointer to derived-class object
invokes base-class print function on that derived-class object:

```

commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04

```

Figura 13.5 Atribuindo endereços de objetos das classes básica e derivada aos ponteiros de classe básica e derivada.

(continuação)

13.3.2 Apontando ponteiros de classe derivada para objetos da classe básica

Na Seção 13.3.1, atribuímos o endereço de um objeto de classe derivada a um ponteiro de classe básica e explicamos que o compilador C++ permite essa atribuição, porque um objeto de classe derivada é um objeto de classe básica. Adotamos a abordagem oposta da Figura 13.6, já que apontamos um ponteiro de classe derivada para um objeto de classe básica. [Nota: Esse programa utiliza as classes `CommissionEmployee` e `BasePlusCommissionEmployee` das figuras 13.1–13.4.] As linhas 8–9 da Figura 13.6 criam um objeto `CommissionEmployee` e a linha 10 cria um ponteiro `BasePlusCommissionEmployee`. A linha 14 tenta atribuir o endereço de objeto de classe básica `commissionEmployee` ao ponteiro de classe derivada `basePlusCommissionEmployeePtr`, mas o compilador C++ gera um erro. O compilador impede essa atribuição, porque um `CommissionEmployee` não é um `BasePlusCommissionEmployee`. Considere as


```

1 // Figura 13.6: fig13_06.cpp
2 // Apontando um ponteiro de classe derivada para um objeto de classe básica.
3 #include "CommissionEmployee.h"
4 #include "BasePlusCommissionEmployee.h"
5
6 int main()
7 {
8     CommissionEmployee commissionEmployee(
9         "Sue", "Jones", "222-22-2222", 10000, .06 );
10    BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;
11
12    // aponta o ponteiro de classe derivada para objeto de classe básica
13    // Erro: um CommissionEmployee não é um BasePlusCommissionEmployee
14    basePlusCommissionEmployeePtr = &commissionEmployee;
15    return 0;
16 } // fim de main

```

Mensagens de erro do compilador de linha de comando Borland C++:

```

Error E2034 Fig13_06\fig13_06.cpp 14: Cannot convert 'CommissionEmployee *'
to 'BasePlusCommissionEmployee *' in function main()

```

Mensagens de erro do compilador GNU C++:

```

fig13_06.cpp:14: error: invalid conversion from `CommissionEmployee*' to
`BasePlusCommissionEmployee*'

```

Mensagens de erro do compilador Microsoft Visual C++.NET:

```

C:\cphttp5_examples\ch13\Fig13_06\fig13_06.cpp(14) : error C2440:
'=' : cannot convert from 'CommissionEmployee *__w64 ' to
'BasePlusCommissionEmployee *'
Cast from base to derived requires dynamic_cast or static_cast

```

Figura 13.6 Apontando um ponteiro de classe derivada para um objeto de classe básica.

consequências se o compilador precisasse permitir essa atribuição. Por meio de um ponteiro `BasePlusCommissionEmployee`, podemos invocar cada função-membro `BasePlusCommissionEmployee`, inclusive `setBaseSalary`, do objeto para o qual o ponteiro aponta (isto é, o objeto de classe básica `commissionEmployee`). Entretanto, o objeto `CommissionEmployee` não fornece uma função-membro `setBaseSalary`, nem um membro de dados `baseSalary` para configurar. Isso poderia resultar em problemas, porque a função-membro `setBaseSalary` iria supor que há um membro de dados `baseSalary` a configurar em sua 'localização normal' em um objeto `BasePlusCommissionEmployee`. Essa memória não pertence ao objeto `CommissionEmployee`, então a função-membro `setBaseSalary` poderia sobrescrever outros dados importantes na memória, possivelmente dados que pertencem a um objeto diferente.

13.3.3 Chamadas de função-membro de classe derivada via ponteiros de classe básica

A partir de um ponteiro de classe básica, o compilador permite invocar somente funções-membro de classes básicas. Portanto, se um ponteiro de classe básica estiver apontado para um objeto de classe derivada e uma tentativa de acessar uma função-membro exclusiva da classe derivada for feita, ocorrerá um erro de compilação.

A Figura 13.7 mostra as consequências de se tentar invocar uma função-membro de classe derivada a partir de um ponteiro de classe básica. [Nota: Estamos utilizando novamente as classes `CommissionEmployee` e `BasePlusCommissionEmployee` das figuras 13.1–13.4.] A linha 9 cria `commissionEmployeePtr` — um ponteiro para um objeto `CommissionEmployee` — e as linhas 10–11 criam um objeto `BasePlusCommissionEmployee`. A linha 14 aponta `commissionEmployeePtr` para o objeto de classe derivada `basePlusCommissionEmployee`. Considerando a Seção 13.3.1, lembre-se de que o compilador C++ permite isso, porque um `BasePlusCommissionEmployee` é um `CommissionEmployee` (no sentido de um objeto `BasePlusCommissionEmployee` contém todas as funcionalidades de um objeto `CommissionEmployee`). As linhas 18–22 invocam as funções-membro de classe básica `getFirstName`, `getLastName`, `getSocialSecurityNumber`, `getGrossSales` e `getCommissionRate` a partir do ponteiro de classe básica. Todas essas chamadas são legítimas, porque

```

1 // Figura 13.7: fig13_07.cpp
2 // Tentando invocar as funções-membro exclusivas da classe derivada
3 // por um ponteiro de classe básica.
4 #include "CommissionEmployee.h"
5 #include "BasePlusCommissionEmployee.h"
6
7 int main()
8 {
9     CommissionEmployee *commissionEmployeePtr = 0; // classe básica
10    BasePlusCommissionEmployee basePlusCommissionEmployee(
11        "Bob", "Lewis", "333-33-3333", 5000, .04, 300 ); // classe derivada
12
13    // aponta o ponteiro de classe básica para o objeto de classe derivada
14    commissionEmployeePtr = &basePlusCommissionEmployee;
15
16    // invoca as funções-membro de classe básica no objeto de classe derivada
17    // por ponteiro de classe básica
18    string firstName = commissionEmployeePtr->getFirstName();
19    string lastName = commissionEmployeePtr->getLastName();
20    string ssn = commissionEmployeePtr->getSocialSecurityNumber();
21    double grossSales = commissionEmployeePtr->getGrossSales();
22    double commissionRate = commissionEmployeePtr->getCommissionRate();
23
24    // tentativa de invocar funções exclusivas de classe derivada
25    // em objeto de classe derivada por meio de um ponteiro de classe básica
26    double baseSalary = commissionEmployeePtr->getBaseSalary();
27    commissionEmployeePtr->setBaseSalary( 500 );
28    return 0;
29 } // fim de main

```

Mensagens de erro do compilador de linha de comando Borland C++:

```

Error E2316 Fig13_07\fig13_07.cpp 26: 'getBaseSalary' is not a member of
'CommissionEmployee' in function main()
Error E2316 Fig13_07\fig13_07.cpp 27: 'setBaseSalary' is not a member of
'CommissionEmployee' in function main()

```

Mensagens de erro do compilador Microsoft Visual C++.NET:

```

C:\cpphttp5_examples\ch13\Fig13_07\fig13_07.cpp(26) : error C2039:
'getBaseSalary' : is not a member of 'CommissionEmployee'
C:\cpphttp5_examples\ch13\Fig13_07\CommissionEmployee.h(10) :
see declaration of 'CommissionEmployee'
C:\cpphttp5_examples\ch13\Fig13_07\fig13_07.cpp(27) : error C2039:
'setBaseSalary' : is not a member of 'CommissionEmployee'
C:\cpphttp5_examples\ch13\Fig13_07\CommissionEmployee.h(10) :
see declaration of 'CommissionEmployee'

```

Mensagens de erro do compilador GNU C++:

```

fig13_07.cpp:26: error: `getBaseSalary' undeclared (first use this function)
fig13_07.cpp:26: error: (Each undeclared identifier is reported only once for
each function it appears in.)
fig13_07.cpp:27: error: `setBaseSalary' undeclared (first use this function)

```

Figura 13.7 Tentando invocar funções-membro exclusivas da classe derivada via um ponteiro da classe básica.

BasePlusCommissionEmployee herda essas funções-membro de CommissionEmployee. Sabemos que commissionEmployeePtr está apontado para um objeto BasePlusCommissionEmployee, então nas linhas 26–27 tentamos invocar as funções-membro BasePlusCommissionEmployee getBaseSalary e setBaseSalary. O compilador C++ gera erros nessas duas linhas, porque elas não são funções-membro da classe básica CommissionEmployee. O handle pode invocar somente aquelas funções que são membros do tipo de classe associada desse handle. (Nesse caso, a partir de um CommissionEmployee *, podemos invocar somente as funções-membro CommissionEmployee setFirstName, getFirstName, setLastName, getLastName, setSocialSecurityNumber, getSocialSecurityNumber, setGrossSales, getGrossSales, setCommissionRate, getCommissionRate, earnings e print.)

Conclui-se que o compilador C++ realmente permite acesso a membros exclusivos de classe derivada a partir de um ponteiro de classe básica que aponta para um objeto de classe derivada se explicitamente fizermos a coerção do ponteiro de classe básica para um ponteiro de classe derivada — uma técnica conhecida como **downcasting**. Como você aprendeu na Seção 13.3.1, é possível apontar um ponteiro de classe básica para um objeto de classe derivada. Entretanto, como demonstramos na Figura 13.7, um ponteiro de classe básica pode ser utilizado para invocar apenas as funções declaradas na classe básica. O downcasting permite a um programa realizar uma operação específica de classe derivada em um objeto de classe derivada apontado por um ponteiro de classe básica. Depois de um downcast, o programa pode invocar funções de classe derivada que não estão na classe básica. Mostraremos um exemplo concreto de downcasting na Seção 13.8.



Observação de engenharia de software 13.3

Se o endereço de um objeto de classe derivada foi atribuído a um ponteiro de uma de suas classes básicas diretas ou indiretas, é aceitável fazer coerção desse ponteiro de classe básica de volta para um ponteiro do tipo da classe derivada. De fato, isso deve ser feito para enviar a esse objeto de classe derivada mensagens que não aparecem na classe básica.

13.3.4 Funções virtuais

Na Seção 13.3.1, apontamos um ponteiro de classe básica CommissionEmployee para um objeto BasePlusCommissionEmployee de classe derivada e, então, invocamos a função-membro print por meio desse ponteiro. Lembre-se de que o tipo do handle determina que funcionalidades da classe invocar. Nesse caso, o ponteiro CommissionEmployee invocou a função-membro print de CommissionEmployee sobre o objeto BasePlusCommissionEmployee, embora o ponteiro estivesse apontado para um objeto BasePlusCommissionEmployee que tem sua própria função print personalizada. *Com as funções virtual, o tipo do objeto apontado, não o tipo do handle, determina qual versão de uma função virtual invocar.*

Primeiro, consideramos as razões pelas quais as funções virtual são úteis. Suponha que um conjunto de classes de formas como Círculo, Triângulo, Retângulo e Quadrado são todas derivadas da classe básica Forma. A capacidade de desenhar a si própria por meio de uma função-membro desenhar poderia ser fornecida a cada uma dessas classes. Embora cada classe tenha sua própria função desenhar, a função para cada forma é bem diferente. Em um programa que desenha um conjunto de formas, seria útil poder tratar todas as formas genericamente como objetos da classe básica Forma. Então, para desenhar qualquer forma, poderíamos simplesmente utilizar um ponteiro da classe básica Forma para invocar a função desenhar e deixar o programa determinar **dinamicamente** (isto é, em tempo de execução) qual função desenhar de classe derivada utilizar, com base no tipo do objeto para o qual o ponteiro da classe básica Forma aponta a qualquer dado momento.

Para permitir esse tipo de comportamento, declaramos desenhar na classe básica como uma **função virtual** e **sobrescrevemos** desenhar em cada uma das classes derivadas para desenhar a forma apropriada. Da perspectiva da implementação, sobrescrever uma função não é diferente de redefinir uma função (que é a abordagem que utilizamos até agora). Uma função sobrescrita em uma classe derivada tem a mesma assinatura e tipo de retorno (isto é, protótipo) que a função que ela sobrescreve em sua classe básica. Se não declararmos a função de classe básica como virtual, podemos redefinir essa função. Por contraste, se declararmos a função de classe básica como virtual, podemos sobrescrever essa função para ativar comportamento polimórfico. Declaramos uma função virtual precedendo o protótipo da função com a palavra-chave virtual na classe básica. Por exemplo,

```
virtual void desenhar() const;
```

apareceria na classe básica Forma. O protótipo precedente declara que a função desenhar é uma função virtual que não aceita argumentos e não retorna nada. A função é declarada const porque, em geral, uma função desenhar não faria alterações no objeto Forma em que ela é invocada. As funções virtuais não têm necessariamente de ser funções const.



Observação de engenharia de software 13.4

Uma vez que uma função é declarada virtual, ela permanece virtual por toda a hierarquia de herança a partir desse ponto, mesmo que essa função não seja declarada explicitamente como virtual quando uma classe a sobrescreve.



Boa prática de programação 13.1

Mesmo que uma função seja implicitamente virtual por causa de uma declaração feita em um ponto mais alto da hierarquia de classes, declare explicitamente essa função virtual em cada nível da hierarquia para promover a clareza do programa.

**Dica de prevenção de erro 13.1**

Quando um programador navega por uma hierarquia de classes para localizar uma classe para reutilização, é possível que uma função nessa classe exiba um comportamento de função virtual mesmo que ele esteja declarado virtual explicitamente. Isso acontece quando a classe herda uma função virtual de sua classe básica e pode produzir erros de lógica sutis. Erros como esses podem ser evitados declarando explicitamente virtual todas as funções virtual por toda a hierarquia de herança.

**Observação de engenharia de software 13.5**

Quando uma classe derivada escolhe não sobrescrever uma função virtual de sua classe básica, a classe derivada simplesmente herda a implementação de função virtual de sua classe básica.

Se o programa invocar uma função virtual por meio de um ponteiro de classe básica para um objeto de classe derivada (por exemplo, `formaPtr->desenhar()`), o programa escolherá a função `desenhar` da classe derivada correta dinamicamente (isto é, em tempo de execução) com base no tipo de objeto — não no tipo de ponteiro. Escolher a função apropriada para a chamada em tempo de execução (em vez de em tempo de compilação) é conhecido como **vinculação dinâmica** ou **vinculação tardia**.

Quando uma função virtual é chamada referenciando um objeto específico por nome e utilizando o operador de seleção de membro (por exemplo, `objetoQuadrado.desenhar()`), a invocação de função é convertida em tempo de compilação (isso é denominado **vinculação estática**) e a função virtual que é chamada é aquela definida para (ou herdada pela) a classe desse objeto particular — esse não é um comportamento polimórfico. Portanto, a vinculação dinâmica com funções virtual ocorre somente a partir de handles de ponteiro (e, como veremos, de referência).

Agora vejamos como as funções virtual podem permitir comportamento polimórfico em nossa hierarquia de empregados. As figuras 13.8–13.9 são os arquivos de cabeçalho para as classes `CommissionEmployee` e `BasePlusCommissionEmployee`, respectivamente. Observe que a única diferença entre esses arquivos e os das figuras 13.1 e 13.3 é que especificamos as funções-membro `earnings` e `print` de cada classe como virtual (linhas 30–31 da Figura 13.8 e linhas 21–22 da Figura 13.9). Como as funções `earnings` e `print` são virtual na classe `CommissionEmployee`, as funções `earnings` e `print` da classe `BasePlusCommissionEmployee` sobrescrevem as da `CommissionEmployee`. Agora, se apontarmos um ponteiro da classe básica `CommissionEmployee` para um objeto da classe derivada `BasePlusCommissionEmployee` e o programa usar esse ponteiro para chamar a função `earnings` ou `print`, a função correspondente do objeto `BasePlusCommissionEmployee` será invocada. Não há nenhuma alteração nas implementações de função-membro das classes `CommissionEmployee` e `BasePlusCommissionEmployee`, então reutilizamos as versões das figuras 13.2 e 13.4.

```

1 // Figura 13.8: CommissionEmployee.h
2 // Classe CommissionEmployee representa um empregado comissionado.
3 #ifndef COMMISSION_H
4 #define COMMISSION_H
5
6 #include <string> // classe string padrão C++
7 using std::string;
8
9 class CommissionEmployee
10 {
11 public:
12     CommissionEmployee( const string &, const string &, const string &,
13                        double = 0.0, double = 0.0 );
14
15     void setFirstName( const string & ); // configura o nome
16     string getFirstName() const; // retorna o nome
17
18     void setLastName( const string & ); // configura o sobrenome
19     string getLastName() const; // retorna o sobrenome
20
21     void setSocialSecurityNumber( const string & ); // configura o SSN
22     string getSocialSecurityNumber() const; // retorna o SSN
23
24     void setGrossSales( double ); // configura a quantidade de vendas brutas
25     double getGrossSales() const; // retorna a quantidade de vendas brutas

```

Figura 13.8 O arquivo de cabeçalho da classe `CommissionEmployee` declara as funções `earnings` e `print` como virtual.

(continua)

```

26
27     void setCommissionRate( double ); // configura a taxa de comissão
28     double getCommissionRate() const; // retorna a taxa de comissão
29
30     virtual double earnings() const; // calcula os rendimentos
31     virtual void print() const; // imprime o objeto CommissionEmployee
32 private:
33     string firstName;
34     string lastName;
35     string socialSecurityNumber;
36     double grossSales; // vendas brutas semanais
37     double commissionRate; // porcentagem da comissão
38 }; // fim da classe CommissionEmployee
39
40 #endif

```

Figura 13.8 O arquivo de cabeçalho da classe `CommissionEmployee` declara as funções `earnings` e `print` como `virtual`. (continuação)

```

1 // Figura 13.9: BasePlusCommissionEmployee.h
2 // Classe BasePlusCommissionEmployee derivada da classe
3 // CommissionEmployee.
4 #ifndef BASEPLUS_H
5 #define BASEPLUS_H
6
7 #include <string> // classe string padrão C++
8 using std::string;
9
10 #include "CommissionEmployee.h" // declaração da classe CommissionEmployee
11
12 class BasePlusCommissionEmployee : public CommissionEmployee
13 {
14 public:
15     BasePlusCommissionEmployee( const string &, const string &,
16                                const string &, double = 0.0, double = 0.0, double = 0.0 );
17
18     void setBaseSalary( double ); // configura o salário-base
19     double getBaseSalary() const; // retorna o salário-base
20
21     virtual double earnings() const; // calcula os rendimentos
22     virtual void print() const; // imprime o objeto BasePlusCommissionEmployee
23 private:
24     double baseSalary; // salário-base
25 }; // fim da classe BasePlusCommissionEmployee
26
27 #endif

```

Figura 13.9 O arquivo de cabeçalho da classe `BasePlusCommissionEmployee` declara as funções `earnings` e `print` como `virtual`.

Modificamos a Figura 13.5 para criar o programa da Figura 13.10. As linhas 46–57 demonstram novamente que um ponteiro `CommissionEmployee` apontado para um objeto `CommissionEmployee` pode ser utilizado para invocar a funcionalidade de `CommissionEmployee`, e um ponteiro `BasePlusCommissionEmployee` apontado para um objeto `BasePlusCommissionEmployee` pode ser utilizado para invocar a funcionalidade de `BasePlusCommissionEmployee`. A linha 60 aponta o ponteiro de classe básica `commissionEmployeePtr` para objeto de classe derivada `basePlusCommissionEmployee`. Observe que, quando a linha 67 invoca a função-membro `print` a partir do ponteiro de classe básica, a função-membro `print` da classe derivada `BasePlusCommissionEmployee` é invocada, então a linha 67 gera saída de um texto diferente do gerado pela linha 59 na Figura 13.5 (quando a função-membro `print` não foi declarada `virtual`).

Vemos que declarar uma função-membro `virtual` faz com que o programa determine dinamicamente que função invocar com base no tipo de objeto para o qual o `handle` aponta, em vez de no tipo do `handle`. A decisão sobre qual função chamar é um exemplo de polimorfismo. Observe novamente que, quando `commissionEmployeePtr` aponta para um objeto `CommissionEmployee` (linha 46), a função `print` da classe `CommissionEmployee` é invocada; e, quando `CommissionEmployeePtr` aponta para um objeto `BasePlusCommissionEmployee`, a função `print` da classe `BasePlusCommissionEmployee` é invocada. Portanto, a mesma mensagem — `print`, nesse caso — enviada (a partir de um ponteiro de classe básica) para uma variedade de objetos relacionados por herança com essa classe básica, assume muitas formas — esse é o comportamento polimórfico.

```

1 // Figura 13.10: fig13_10.cpp
2 // Introduzindo polimorfismo, funções virtual e vinculação dinâmica.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::fixed;
7
8 #include <iomanip>
9 using std::setprecision;
10
11 // inclui definições de classe
12 #include "CommissionEmployee.h"
13 #include "BasePlusCommissionEmployee.h"
14
15 int main()
16 {
17     // cria objeto de classe básica
18     CommissionEmployee commissionEmployee(
19         "Sue", "Jones", "222-22-2222", 10000, .06 );
20
21     // cria ponteiro de classe básica
22     CommissionEmployee *commissionEmployeePtr = 0;
23
24     // cria objeto de classe derivada
25     BasePlusCommissionEmployee basePlusCommissionEmployee(
26         "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );
27
28     // cria ponteiro de classe derivada
29     BasePlusCommissionEmployee *basePlusCommissionEmployeePtr = 0;
30
31     // configura a formatação de saída de ponto flutuante
32     cout << fixed << setprecision( 2 );
33
34     // gera saída de objetos utilizando vinculação estática
35     cout << "Invoking print function on base-class and derived-class "
36         << "\nobjects with static binding\n\n";
37     commissionEmployee.print(); // vinculação estática
38     cout << "\n\n";
39     basePlusCommissionEmployee.print(); // vinculação estática
40
41     // gera saída de objetos utilizando vinculação dinâmica
42     cout << "\n\n\nInvoking print function on base-class and "
43         << "derived-class \nobjects with dynamic binding";
44
45     // aponta o ponteiro de classe básica para o objeto de classe básica e imprime
46     commissionEmployeePtr = &commissionEmployee;

```

Figura 13.10 Demonstrando polimorfismo invocando uma função `virtual` de classe derivada via um ponteiro de classe básica para um objeto de classe derivada. (continua)

```

47     cout << "\n\nCalling virtual function print with base-class pointer"
48         << "\nto base-class object invokes base-class "
49         << "print function:\n\n";
50     commissionEmployeePtr->print(); // invoca print da classe básica
51
52     // aponta o ponteiro de classe derivada para o objeto de classe derivada e imprime
53     basePlusCommissionEmployeePtr = &basePlusCommissionEmployee;
54     cout << "\n\nCalling virtual function print with derived-class "
55         << "pointer\nto derived-class object invokes derived-class "
56         << "print function:\n\n";
57     basePlusCommissionEmployeePtr->print(); // invoca print da classe derivada
58
59     // aponta o ponteiro de classe básica para o objeto de classe derivada e imprime
60     commissionEmployeePtr = &basePlusCommissionEmployee;
61     cout << "\n\nCalling virtual function print with base-class pointer"
62         << "\nto derived-class object invokes derived-class "
63         << "print function:\n\n";
64
65     // polimorfismo; invoca print de BasePlusCommissionEmployee;
66     // ponteiro de classe básica para objeto de classe derivada
67     commissionEmployeePtr->print();
68     cout << endl;
69     return 0;
70 } // fim de main

```

Invoking print function on base-class and derived-class objects with static binding

```

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

```

```

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00

```

Invoking print function on base-class and derived-class objects with dynamic binding

Calling virtual function print with base-class pointer
to base-class object invokes base-class print function:

```

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

```

Calling virtual function print with derived-class pointer
to derived-class object invokes derived-class print function:

```

base-salaried commission employee: Bob Lewis

```

Figura 13.10 Demonstrando polimorfismo invocando uma função virtual de classe derivada via um ponteiro de classe básica para um objeto de classe derivada. (continua)


```

social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00

Calling virtual function print with base-class pointer
to derived-class object invokes derived-class print function:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00

```

Figura 13.10 Demonstrando polimorfismo invocando uma função virtual de classe derivada via um ponteiro de classe básica para um objeto de classe derivada. (continuação)

13.3.5 Resumo das atribuições permitidas entre objetos de classe básica e de classe derivada e ponteiros

Agora que você viu um aplicativo completo que processa diversos objetos polimorficamente, resumimos o que você pode e o que você não pode fazer com objetos e ponteiros de classe básica e derivada. Embora um objeto da classe derivada também seja um objeto de classe básica, os dois objetos são, apesar disso, diferentes. Como discutido anteriormente, os objetos de classe derivada podem ser tratados como se fossem objetos de classe básica. Esse é um relacionamento lógico, porque a classe derivada contém todos os membros da classe básica. Entretanto, os objetos de classe básica não podem ser tratados como se fossem de classe derivada — a classe derivada pode ter membros exclusivos de classes derivadas adicionais. Por essa razão, não é permitido apontar um ponteiro de classe derivada para um objeto de classe básica sem uma coerção explícita — essa atribuição deixaria os membros exclusivos da classe derivada indefinidos no objeto de classe básica. A coerção alivia o compilador da responsabilidade de emitir uma mensagem de erro. De certo modo, utilizando a coerção você está dizendo: ‘sei que o que estou fazendo é perigoso e assumo toda a responsabilidade por minhas ações’.

Na seção atual e no Capítulo 12, discutimos quatro maneiras de apontar ponteiros de classe básica e ponteiros de classe derivada para objetos de classe básica e objetos de classe derivada:

1. Apontar um ponteiro de classe básica para um objeto de classe básica é simples e direto — as chamadas feitas a partir do ponteiro de classe básica simplesmente invocam as funcionalidades da classe básica.
2. Apontar um ponteiro de classe derivada para um objeto de classe derivada é simples e direto — as chamadas feitas a partir do ponteiro de classe derivada simplesmente invocam as funcionalidades de classe derivada.
3. É seguro apontar um ponteiro de classe básica para um objeto de classe derivada, porque o objeto de classe derivada *é um* objeto de sua classe básica. Entretanto, esse ponteiro pode ser utilizado para invocar apenas as funções-membro da classe básica. Se o programador tentar referenciar um membro exclusivo da classe derivada por meio do ponteiro de classe básica, o compilador informa um erro. Para evitar esse erro, o programador deve fazer coerção do ponteiro de classe básica para um ponteiro de classe derivada. O ponteiro de classe derivada então pode ser utilizado para invocar as funcionalidades completas do objeto de classe derivada. Entretanto, essa técnica — chamada de downcasting — é uma operação potencialmente perigosa. A Seção 13.8 demonstra como utilizar downcasting com segurança.
4. Apontar um ponteiro de classe derivada para um objeto de classe básica gera um erro de compilação. O relacionamento *é um* se aplica apenas de uma classe derivada para suas classes básicas direta e indireta, e não vice-versa. Um objeto de classe básica não contém membros exclusivos de classe derivada que podem ser invocados a partir de um ponteiro de classe derivada.



Erro comum de programação 13.1

Depois de apontar um ponteiro de classe básica para um objeto de classe derivada, tentar referenciar membros exclusivos de classe derivada com o ponteiro de classe básica é um erro de compilação.



Erro comum de programação 13.2

Tratar um objeto de classe básica como um objeto de classe derivada pode causar erros.

13.4 Campos de tipo e instruções switch

Uma maneira de determinar o tipo de um objeto que é incorporado em um programa maior é utilizar uma instrução `switch`. Isso permite distinguir entre os tipos de objeto e, então, invocar uma ação apropriada para um objeto particular. Por exemplo, em uma hierarquia de formas em que cada objeto forma tem um atributo `tipoDeForma`, uma instrução `switch` poderia verificar o `tipoDeForma` do objeto para determinar qual função imprimir chamar.

Entretanto, utilizar a lógica `switch` expõe os programas a uma variedade de problemas potenciais. Por exemplo, o programador poderia se esquecer de incluir um teste de tipo quando um é garantido ou poderia se esquecer de testar todos os casos possíveis em uma instrução `switch`. Ao modificar um sistema baseado em `switch` adicionando novos tipos, o programador poderia se esquecer de inserir os novos casos em todas as instruções `switch` relevantes. Cada adição ou exclusão de uma classe requer a modificação de todas as instruções `switch` no sistema; a monitoração dessas instruções pode ser demorada e propensa a erro.



Observação de engenharia de software 13.6

A programação polimórfica pode eliminar a necessidade de lógica `switch` desnecessária. Usando o mecanismo de polimorfismo do C++ para realizar uma lógica equivalente, os programadores podem evitar os tipos de erro normalmente associados com a lógica `switch`.



Observação de engenharia de software 13.7

Uma consequência interessante de utilizar polimorfismo é que os programas assumem uma aparência simplificada. Eles contêm menos lógica de desvio e código mais simples, sequencial. Essa simplificação facilita o teste, depuração e manutenção do programa.

13.5 Classes abstratas e funções virtual puras

Quando pensamos em classe como um tipo, supomos que os programas criam objetos desse tipo. Entretanto, há casos em que é útil definir as classes a partir das quais o programador nunca pretenderá instanciar qualquer objeto. Essas classes são chamadas **classes abstratas**. Como essas classes são normalmente utilizadas como classes básicas em hierarquias de herança, elas são referidas como **classes básicas abstratas**. Essas classes não podem ser utilizadas para instanciar objetos, porque, como veremos em breve, as classes abstratas são incompletas — as classes derivadas devem definir as ‘partes ausentes’. Construímos programas com classes abstratas na Seção 13.6.

O propósito de uma classe abstrata é fornecer uma classe básica apropriada a partir da qual outras classes podem herdar. As classes que podem ser utilizadas para instanciar objetos são chamadas **classes concretas**. Essas classes fornecem implementações de cada função-membro que elas definem. Poderíamos ter uma classe básica abstrata `FormaBidimensional` e derivar tais classes concretas como `Quadrado`, `Círculo` e `Triângulo`. Também poderíamos ter uma classe básica abstrata `FormaTridimensional` e derivar essas classes concretas como `Cubo`, `Esfera` e `Cilindro`. As classes básicas abstratas são muito genéricas para definir objetos reais; precisamos ser mais específicos antes de podermos pensar em instanciar objetos. Por exemplo, se alguém lhe disser para ‘desenhar a forma bidimensional’, que forma você desenharia? As classes concretas fornecem os aspectos específicos que tornam razoável instanciar objetos.

Uma hierarquia de herança não precisa conter nenhuma classe abstrata, mas, como veremos, muitos bons sistemas orientados a objetos têm hierarquias de classe encabeçadas por classes básicas abstratas. Em alguns casos, as classes abstratas constituem os poucos níveis superiores da hierarquia. Um bom exemplo disso é a hierarquia de formas na Figura 12.3, que começa com a classe básica abstrata `Forma`. No próximo nível da hierarquia temos mais duas classes básicas abstratas, a saber, `FormaBidimensional` e `FormaTridimensional`. O próximo nível da hierarquia define classes concretas para formas bidimensionais (a saber, `Círculo`, `Quadrado` e `Triângulo`) e para formas tridimensionais (a saber, `Esfera`, `Cubo` e `Tetraedro`).

Tornamos uma classe abstrata declarando uma ou mais de suas funções `virtual` como ‘puras’. Uma **função virtual pura** é especificada colocando-se ‘= 0’ em sua declaração, como em

```
virtual void draw() const = 0; // função virtual pura
```

O ‘= 0’ é conhecido como um **especificador puro**. As funções `virtual` puras não fornecem implementações. Toda classe derivada concreta deve sobrescrever todas as funções `virtual` puras da classe básica com implementações concretas dessas funções. A diferença entre uma função `virtual` e uma função `virtual` pura é que uma função `virtual` tem uma implementação e dá à classe derivada a opção de sobrescrever a função; em contraste, uma função `virtual` pura não fornece uma implementação e requer que a classe derivada sobrescreva a função (para que classe derivada seja concreta; caso contrário, a classe derivada permanece abstrata).

As funções `virtual` puras são utilizadas quando não faz sentido a classe básica ter uma implementação de uma função, mas o programador quer que todas as classes derivadas concretas implementem a função. Retornando ao nosso exemplo anterior de objetos espaciais, não faz sentido a classe básica `ObjetoEspacial` ter uma implementação para a função `desenhar` (já que não há nenhuma maneira de desenhar um objeto espacial genérico sem ter as informações adicionais sobre que tipo de objeto espacial está sendo desenhado). Um exemplo de uma função que seria definida como `virtual` (e não `virtual` pura) seria uma que retornasse um nome para o objeto. Podemos nomear um `ObjetoEspacial` genérico (por exemplo, como ‘objeto espacial’), assim, uma implementação-padrão para essa função pode ser fornecida e a função não precisa ser `virtual` pura. Entretanto, a função ainda é declarada `virtual`, porque se espera que as classes derivadas sobrescrevam essa função para fornecer nomes mais específicos para os objetos de classe derivada.

**Observação de engenharia de software 13.8**

Uma classe abstrata define uma interface pública comum para as várias classes em uma hierarquia de classes. Uma classe abstrata contém uma ou mais funções virtual puras que as classes derivadas concretas devem sobrescrever.

**Erro comum de programação 13.3**

Tentar instanciar um objeto de uma classe abstrata causa um erro de compilação.

**Erro comum de programação 13.4**

A falha em sobrescrever uma função virtual pura em uma classe derivada, e então tentar instanciar objetos dessa classe, é um erro de compilação.

**Observação de engenharia de software 13.9**

Uma classe abstrata tem pelo menos uma função virtual pura. Uma classe abstrata também pode ter membros de dados e funções concretas (incluindo construtores e destrutores), que estão sujeitos às regras normais de herança por classes derivadas.

Embora não possamos instanciar objetos de uma classe básica abstrata, podemos utilizar a classe básica abstrata para declarar ponteiros e referências que podem referenciar objetos de qualquer classe concreta derivada da classe abstrata. Em geral, os programas utilizam esses ponteiros e referências para manipular objetos de classe derivada polimorficamente.

Consideremos outra aplicação de polimorfismo. Um gerenciador de tela precisa exibir uma variedade de objetos, inclusive os novos tipos de objetos que o programador adicionará ao sistema depois de escrever o gerenciador de tela. O sistema pode precisar exibir várias formas, como Círculos, Triângulos ou Retângulos, que são derivadas da classe básica abstrata Forma. O gerenciador de tela utiliza ponteiros Forma para gerenciar os objetos que são exibidos. Para desenhar qualquer objeto (independentemente do nível em que a classe desse objeto aparece na hierarquia de herança), o gerenciador de tela usa um ponteiro de classe básica para o objeto invocar a função desenhar do objeto, que é uma função virtual pura na classe básica Forma; portanto, cada classe derivada concreta deve implementar a função desenhar. Cada objeto Forma na hierarquia de herança tem a capacidade de desenhar a si mesmo. O gerenciador de tela não precisa se preocupar com o tipo de cada objeto, nem se o gerenciador de tela encontrou ou não alguma vez objetos desse tipo.

O polimorfismo é particularmente eficaz para implementar sistemas em camadas de software. Em sistemas operacionais, por exemplo, cada tipo de dispositivo físico poderia operar diferentemente dos outros. Mesmo assim, os comandos para ler (read) ou gravar (write) os dados de e a partir de dispositivos poderiam ter certa uniformidade. A mensagem de gravar enviada para um objeto driver de dispositivo precisa ser interpretada especificamente no contexto desse driver de dispositivo e como esse driver de dispositivo manipula dispositivos de um tipo específico. Entretanto, a própria chamada gravar não é realmente diferente de gravar em qualquer outro dispositivo no sistema — simplesmente transfira um número de bytes da memória para esse dispositivo. Um sistema operacional orientado a objetos talvez utilize uma classe básica abstrata para fornecer uma interface apropriada a todos os drivers de dispositivo. Então, por meio da herança dessa classe básica abstrata, todas as classes derivadas que são formadas operam de maneira semelhante. As capacidades (isto é, as funções public) oferecidas pelos drivers de dispositivo são fornecidas como funções virtual puras na classe básica abstrata. As implementações dessas funções virtual puras são fornecidas nas classes derivadas que correspondem aos tipos específicos de drivers de dispositivo. Essa arquitetura também permite que os novos dispositivos sejam facilmente adicionados a um sistema, mesmo depois de o sistema operacional ter sido definido. O usuário pode simplesmente conectar o dispositivo e instalar seu novo driver de dispositivo. O sistema operacional ‘conversa’ com esse novo dispositivo por meio de seu driver de dispositivo, que tem as mesmas funções-membro public que todos os outros drivers de dispositivo — aqueles definidos na classe básica abstrata de driver de dispositivo.

É comum em programação orientada a objetos definir uma classe iterador que pode percorrer todos os objetos em um contêiner (como um array). Por exemplo, um programa pode imprimir uma lista de objetos em um vector criando um objeto iterador e, então, utilizar o iterador para obter o próximo elemento da lista toda vez que o iterador for chamado. Os iteradores são freqüentemente usados na programação polimórfica para percorrer um array ou uma lista vinculada de ponteiros para objetos de vários níveis de uma hierarquia. Os ponteiros em uma lista dessas são todos ponteiros de classe básica. (O Capítulo 23, “Standard Template Library (STL)”, apresenta um tratamento completo de iteradores.) Uma lista de ponteiros para objetos da classe básica FormaBidimensional poderia conter ponteiros para objetos das classes Quadrado, Círculo, Triângulo e assim por diante. Utilizar o polimorfismo para enviar uma mensagem desenhar, a partir de um ponteiro FormaBidimensional *, para cada objeto na lista desenharia cada objeto corretamente na tela.

13.6 Estudo de caso: sistema de folha de pagamento utilizando polimorfismo

Esta seção reexamina a hierarquia CommissionEmployee–BasePlusCommissionEmployee que exploramos integralmente na Seção 12.4. Neste exemplo, utilizamos uma classe abstrata e polimorfismo para realizar cálculos de folha de pagamento baseados no tipo de empregado. Criamos uma hierarquia de funcionários aprimorada para resolver o problema a seguir:

Uma empresa paga seus empregados semanalmente. Os funcionários são de quatro tipos: funcionários assalariados que recebem salários fixos semanais independentemente do número de horas trabalhadas, funcionários que trabalham por hora e são pagos da mesma forma e recebem horas extras por todas as horas trabalhadas além das 40 horas normais, funcionários comissionados que recebem uma porcentagem sobre suas vendas e funcionários assalariados-comissionados que recebem um salário-base mais uma porcentagem sobre suas vendas. Para o período de pagamento atual, a empresa decidiu recompensar os empregados comissionados assalariados adicionando 10 por cento ao salário-base. A empresa quer implementar um programa C++ que realiza o pagamento polimorficamente na folha de pagamento.

Utilizamos a classe abstrata `Employee` para representar o conceito geral de um funcionário. As classes que derivam diretamente de `Employee` são `SalariedEmployee`, `CommissionEmployee` e `HourlyEmployee`. A classe `BasePlusCommissionEmployee` — derivada de `CommissionEmployee` — representa o último tipo de empregado. O diagrama de classes UML na Figura 13.11 mostra a hierarquia de herança do nosso aplicativo polimórfico de folha de pagamento de funcionários. Observe que o nome da classe abstrata `Employee` é escrito em itálico, de acordo com a convenção da UML.

A classe básica abstrata `Employee` declara a ‘interface’ para a hierarquia — isto é, o conjunto de funções-membro que um programa pode invocar em todos os objetos `Employee`. Cada funcionário, independentemente de como seus vencimentos são calculados, tem um nome, um sobrenome e um número de seguro social, portanto os membros de dados `private firstName`, `lastName` e `socialSecurityNumber` aparecem na classe de base abstrata `Employee`.



Observação de engenharia de software 13.10

Uma classe derivada pode herdar a interface ou implementação de uma classe básica. As hierarquias projetadas para a **herança de implementação** tendem a ter suas funcionalidades na parte superior da hierarquia — cada nova classe derivada herda uma ou mais funções-membro que foram definidas em uma classe básica e a classe derivada utiliza as definições de classe básica. As hierarquias projetadas para a **herança de interface** tendem a ter suas funcionalidades na parte inferior da hierarquia — uma classe básica especifica uma ou mais funções que devem ser definidas para cada classe na hierarquia (isto é, elas têm o mesmo protótipo), mas as classes derivadas individuais fornecem suas próprias implementações da(s) função(ões).

As seções a seguir implementam a hierarquia da classe `Employee`. Cada uma das cinco primeiras implementa uma das classes abstratas ou concretas. A última seção implementa um programa de teste que constrói objetos de todas essas classes e processa os objetos polimorficamente.

13.6.1 Criando a classe básica abstrata `Employee`

A classe `Employee` (figuras 13.13–13.14, discutidas em mais detalhes em breve) fornece as funções `earnings` e `print`, além das várias funções `get` e `set` que manipulam membros de dados da classe `Employee`. Uma função `earnings` certamente se aplica genericamente a todos os empregados, mas todos os cálculos de rendimentos dependem da classe do empregado. Portanto, declaramos `earnings` como `virtual` pura na classe básica `Employee`, porque uma implementação-padrão não faz sentido para essa função — não há informações suficientes para determinar que quantia `earnings` deve retornar. Toda classe derivada sobrescreve `earnings` com uma implementação apropriada. Para calcular os rendimentos de um empregado, o programa atribui o endereço do objeto de um empregado a um ponteiro `Employee` da classe básica e, então, invoca a função `earnings` nesse objeto. Mantemos um vector de ponteiros `Employee`, cada um dos quais aponta para um objeto `Employee` (naturalmente, não pode haver objetos `Employee`, porque `Employee` é uma classe abstrata — por causa da herança, porém, todos os objetos de todas as classes derivadas de `Employee` podem, apesar disso, ser considerados objetos `Employee`). O programa itera pelo vector e chama a função `earnings` para cada objeto `Employee`. O C++ processa essas chamadas de função polimorficamente. Incluir `earnings` como uma função `virtual` pura em `Employee` força cada classe derivada direta de `Employee` que deseja ser uma classe concreta a sobrescrever `earnings`. Isso permite ao projetista da hierarquia de classes exigir que cada classe derivada forneça um cálculo de salário apropriado, se de fato essa classe derivada precisar ser concreta.

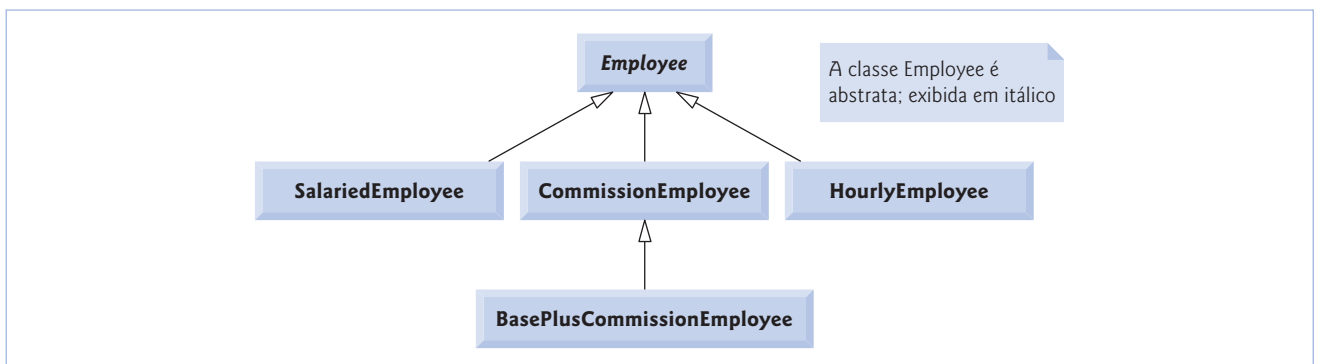


Figura 13.11 Diagrama de classes UML da hierarquia `Employee`.

A função `print` na classe `Employee` exibe o nome, o sobrenome e o número de seguro social do empregado. Como veremos, cada classe derivada de `Employee` sobrescreve a função `print` para gerar saída do tipo do empregado (por exemplo, "salaried employee:") seguido pelas demais informações do empregado.

O diagrama na Figura 13.12 mostra cada uma das cinco classes da hierarquia no lado esquerdo e as funções `earnings` e `print` na parte superior. Para cada classe, o diagrama mostra os resultados desejados de cada função. Observe que a classe `Employee` especifica `= 0` para a função `earnings` indicar que essa é uma função `virtual` pura. Cada classe derivada sobrescreve essa função para fornecer uma implementação apropriada. Não listamos as funções `get` e `set` da classe básica `Employee` porque elas não são sobrescritas em qualquer classe derivada — cada uma dessas funções é herdada e utilizada 'como é' por cada uma das classes derivadas.

Consideremos o arquivo de cabeçalho da classe `Employee` (Figura 13.13). As funções-membro `public` incluem um construtor que aceita o nome, o sobrenome e o número de seguro social como argumentos (linha 12); as funções `set` que configuram o nome, o sobrenome e o número de seguro social (linhas 14, 17 e 20, respectivamente); as funções `get` que retornam o nome, o sobrenome e o número de seguro social (linhas 15, 18 e 21, respectivamente); a função `virtual earnings` pura (linha 24) e a função `virtual print` (linha 25).

Lembre-se de que declaramos `earnings` como uma função `virtual` pura porque primeiro *devemos* conhecer o tipo de `Employee` específico para determinar os cálculos de `earnings` apropriados. Declarar essa função como `virtual` pura indica que cada classe derivada concreta deve fornecer uma implementação de `earnings` apropriada e que um programa pode utilizar os ponteiros de classe básica `Employee` para invocar a função `earnings` polimorficamente para qualquer tipo de `Employee`.

A Figura 13.14 contém as implementações de função-membro para a classe `Employee`. Nenhuma implementação é oferecida para a função `earnings virtual`. Observe que o construtor `Employee` (linhas 10–15) não valida o número do seguro social. Normalmente, essa validação deve ser fornecida. Um exercício do Capítulo 12 pede para você validar um número de seguro social a fim de assegurar que ele esteja na forma `###-##-####`, onde cada `#` representa um dígito.

Observe que a função `virtual print` (Figura 13.14, linhas 54–58) fornece uma implementação que será sobrescrita em cada uma das classes derivadas. Mas cada uma dessas funções utilizará a versão de `print` da classe abstrata para imprimir informações comuns a todas as classes na hierarquia `Employee`.

13.6.2 Criando a classe derivada concreta `SalariedEmployee`

A classe `SalariedEmployee` (figuras 13.15–13.16) deriva da classe `Employee` (linha 8 da Figura 13.15). As funções-membro `public` incluem um construtor que aceita um nome, um sobrenome, um número de seguro social e um salário semanal como argumentos (linhas 11–12); uma função `set` para atribuir um novo valor não negativo ao membro de dados `weeklySalary` (linha 14); uma função `get` para retornar o valor de `weeklySalary` (linha 15); uma função `virtual earnings` que calcula os rendimentos de um `SalariedEmployee` (linha 18) e uma função `virtual print` que gera saída do tipo do empregado, a saber, "salaried employee:" seguido por informações específicas do empregado produzidas pela função `print` da classe básica `Employee` e pela função `getWeeklySalary` de `SalariedEmployee` (linha 19).

	earnings	print
Employee	<code>= 0</code>	<code>firstName lastName</code> <code>social security number: SSN</code>
Salaried- Employee	<code>weeklySalary</code>	<code>salaried employee: firstName lastName</code> <code>social security number: SSN</code> <code>weekly salary: weeklysalar</code>
Hourly- Employee	<code>If hours <= 40</code> <code> wage * hours</code> <code>If hours > 40</code> <code> (40 * wage) + ((hours - 40) * wage * 1.5)</code>	<code>hourly employee: firstName lastName</code> <code>social security number: SSN</code> <code>hourly wage: wage; hours worked: hours</code>
Commission- Employee	<code>commissionRate * grossSales</code>	<code>commission employee: firstName lastName</code> <code>social security number: SSN</code> <code>gross sales: grossSales;</code> <code>commission rate: commissionRate</code>
BasePlus- Commission- Employee	<code>baseSalary + (commissionRate * grossSales)</code>	<code>base salaried commission employee: firstName lastName</code> <code>social security number: SSN</code> <code>gross sales: grossSales;</code> <code>commission rate: commissionRate;</code> <code>base salary: baseSalary</code>

Figura 13.12 Interface polimórfica para as classes na hierarquia `Employee`.


```

1 // Figura 13.13: Employee.h
2 // Classe básica abstrata Employee.
3 #ifndef EMPLOYEE_H
4 #define EMPLOYEE_H
5
6 #include <string> // classe string padrão C++
7 using std::string;
8
9 class Employee
10 {
11 public:
12     Employee( const string &, const string &, const string & );
13
14     void setFirstName( const string & ); // configura o nome
15     string getFirstName() const; // retorna o nome
16
17     void setLastName( const string & ); // configura o sobrenome
18     string getLastName() const; // retorna o sobrenome
19
20     void setSocialSecurityNumber( const string & ); // configura o SSN
21     string getSocialSecurityNumber() const; // retorna o SSN
22
23     // a função virtual pura cria a classe básica abstrata Employee
24     virtual double earnings() const = 0; // virtual pura
25     virtual void print() const; // virtual
26 private:
27     string firstName;
28     string lastName;
29     string socialSecurityNumber;
30 }; // fim da classe Employee
31
32 #endif // EMPLOYEE_H

```

Figura 13.13 Arquivo de cabeçalho da classe Employee.

```

1 // Figura 13.14: Employee.cpp
2 // Definições de função-membro da classe básica abstrata Employee.
3 // Nota: Nenhuma definição recebe funções virtuais puras.
4 #include <iostream>
5 using std::cout;
6
7 #include "Employee.h" // Definição da classe Employee
8
9 // construtor
10 Employee::Employee( const string &first, const string &last,
11     const string &ssn )
12     : firstName( first ), lastName( last ), socialSecurityNumber( ssn )
13 {
14     // corpo vazio
15 } // fim do construtor Employee
16
17 // configura o nome
18 void Employee::setFirstName( const string &first )
19 {

```

Figura 13.14 Arquivo de implementação da classe Employee.

(continua)

```

20     firstName = first;
21 } // fim da função setFirstName
22
23 // retorna o nome
24 string Employee::getFirstName() const
25 {
26     return firstName;
27 } // fim da função getFirstName
28
29 // configura o sobrenome
30 void Employee::setLastName( const string &last )
31 {
32     lastName = last;
33 } // fim da função setLastName
34
35 // retorna o sobrenome
36 string Employee::getLastName() const
37 {
38     return lastName;
39 } // fim da função getLastName
40
41 // configura o SSN
42 void Employee::setSocialSecurityNumber( const string &ssn )
43 {
44     socialSecurityNumber = ssn; // deve validar
45 } // fim da função setSocialSecurityNumber
46
47 // retorna o SSN
48 string Employee::getSocialSecurityNumber() const
49 {
50     return socialSecurityNumber;
51 } // fim da função getSocialSecurityNumber
52
53 // imprime informações de Employee (virtual, mas não virtual pura)
54 void Employee::print() const
55 {
56     cout << getFirstName() << ' ' << getLastName()
57         << "\nsocial security number: " << getSocialSecurityNumber();
58 } // fim da função print

```

Figura 13.14 Arquivo de implementação da classe Employee.

(continuação)

```

1 // Figura 13.15: SalariedEmployee.h
2 // Classe SalariedEmployee derivada de Employee.
3 #ifndef SALARIED_H
4 #define SALARIED_H
5
6 #include "Employee.h" // definição da classe Employee
7
8 class SalariedEmployee : public Employee
9 {
10 public:
11     SalariedEmployee( const string &, const string &,
12                     const string &, double = 0.0 );

```

Figura 13.15 Arquivo de cabeçalho da classe SalariedEmployee.

(continua)


```

13
14     void setWeeklySalary( double ); // configura o salário semanal
15     double getWeeklySalary() const; // retorna o salário semanal
16
17     // palavra-chave virtual assinala intenção de sobrescrever
18     virtual double earnings() const; // calcula os rendimentos
19     virtual void print() const; // imprime objeto SalariedEmployee
20 private:
21     double weeklySalary; // salário por semana
22 }; // fim da classe SalariedEmployee
23
24 #endif // SALARIED_H

```

Figura 13.15 Arquivo de cabeçalho da classe SalariedEmployee.

(continuação)

A Figura 13.16 contém as implementações de função-membro de SalariedEmployee. O construtor da classe passa o nome, o sobrenome e o número de seguro social para o construtor Employee (linha 11) a fim de inicializar os membros de dados private que são herdados da classe básica, mas não acessíveis na classe derivada. A função earnings (linha 30–33) sobrescreve a função virtual earnings pura em Employee para fornecer uma implementação concreta que retorna o salário semanal de SalariedEmployee. Se não implementássemos earnings, a classe SalariedEmployee seria uma classe abstrata e qualquer tentativa de instanciar um objeto da classe resultaria em um erro de compilação (e, naturalmente, queremos que SalariedEmployee aqui seja uma classe concreta). Observe que no arquivo de cabeçalho da classe SalariedEmployee, declaramos as funções-membro earnings e print como virtual (linhas 18–19 da Figura 13.15) — na realidade, é redundante colocar a palavra-chave virtual antes dessas funções-membro. Definimos essas funções como virtual na classe básica Employee, então elas permanecem funções virtual por toda a hierarquia de classes. Considerando a “Boa prática de programação 13.1”, lembre-se de que declarar tais funções virtual explicitamente em cada nível da hierarquia pode promover clareza de programa.

A função print da classe SalariedEmployee (linhas 36–41 da Figura 13.16) sobrescreve a função print Employee. Se a classe SalariedEmployee não sobrescrevesse print, SalariedEmployee herdaria a versão Employee de print. Nesse caso, a função print de SalariedEmployee simplesmente retornaria o nome completo e o número de seguro social do funcionário, o que não representa adequadamente um SalariedEmployee. Para imprimir informações completas de um SalariedEmployee, a função print da classe derivada gera a saída de "salaried employee: " seguida pelas informações específicas da classe básica Employee (isto é, o nome, o sobrenome e o número de seguro social) impressas invocando print da classe básica utilizando o operador de resolução de escopo (linha 39) — esse é um exemplo elegante de reutilização de código. A saída produzida pela função print de SalariedEmployee contém o salário semanal do empregado obtido invocando a função getWeeklySalary da classe.

```

1 // Figura 13.16: SalariedEmployee.cpp
2 // Definições de função-membro da classe SalariedEmployee.
3 #include <iostream>
4 using std::cout;
5
6 #include "SalariedEmployee.h" // definição da classe SalariedEmployee
7
8 // construtor
9 SalariedEmployee::SalariedEmployee( const string &first,
10     const string &last, const string &ssn, double salary )
11     : Employee( first, last, ssn )
12 {
13     setWeeklySalary( salary );
14 } // fim do construtor SalariedEmployee
15
16 // configura o salário
17 void SalariedEmployee::setWeeklySalary( double salary )

```

Figura 13.16 Arquivo de implementação da classe SalariedEmployee.

(continua)

```

18 {
19     weeklySalary = ( salary < 0.0 ) ? 0.0 : salary;
20 } // fim da função setWeeklySalary
21
22 // retorna o salário
23 double SalariedEmployee::getWeeklySalary() const
24 {
25     return weeklySalary;
26 } // fim da função getWeeklySalary
27
28 // calcula os rendimentos;
29 // sobrescreve a função virtual pura earnings em Employee
30 double SalariedEmployee::earnings() const
31 {
32     return getWeeklySalary();
33 } // fim da função earnings
34
35 // imprime informações de SalariedEmployee
36 void SalariedEmployee::print() const
37 {
38     cout << "salaried employee: ";
39     Employee::print(); // reutiliza função print da classe básica abstrata
40     cout << "\nweekly salary: " << getWeeklySalary();
41 } // fim da função print

```

Figura 13.16 Arquivo de implementação da classe `SalariedEmployee`.

(continuação)

13.6.3 Criando a classe derivada concreta `HourlyEmployee`

A classe `HourlyEmployee` (figuras 13.17–13.18) também deriva da classe `Employee` (linha 8 da Figura 13.17). As funções-membro `public` incluem um construtor (linhas 11–12) que aceita como argumentos um nome, um sobrenome, um número de seguro social, um salário-hora e o número de horas trabalhadas; as funções `set` que atribuem novos valores aos membros de dados `wage` e `hours`, respectivamente (linhas 14 e 17); as funções `get` para retornar os valores de `wage` e `hours`, respectivamente (linhas 15 e 18); uma função virtual `earnings` que calcula os rendimentos de um `HourlyEmployee` (linha 21) e uma função virtual `print` que gera saída do tipo do empregado, a saber, "hourly employee: " e informações específicas do empregado (linha 22).

A Figura 13.18 contém as implementações de função-membro para a classe `HourlyEmployee`. As linhas 18–21 e 30–34 definem as funções `set` que atribuem novos valores aos membros de dados `wage` e `hours`, respectivamente. A função `setWage` (linhas 18–21) assegura que `wage` é não negativo, e a função `setHours` (linhas 30–34) assegura que o membro de dados `hours` está entre 0 e 168 (o número total de horas em uma semana). As funções `get` da classe `HourlyEmployee` são implementadas nas linhas 24–27 e 37–40. Não declaramos essas funções `virtual`, portanto as classes derivadas da classe `HourlyEmployee` não podem sobrescrevê-las (embora as classes derivadas certamente possam redefini-las). Observe que o construtor `HourlyEmployee`, como o construtor `SalariedEmployee`, passa o nome, o sobrenome e o número de seguro social ao construtor `Employee` da classe básica (linha 11) para inicializar os membros de dados `private` herdados declarados na classe básica. Além disso, a função `print` de `HourlyEmployee` chama a função `print` da classe básica (linha 56) para gerar saída de informações específicas do `Employee` (isto é, o nome, o sobrenome e o número de seguro social) — esse é outro exemplo elegante de reutilização de código.

13.6.4 Criando a classe derivada concreta `CommissionEmployee`

A classe `CommissionEmployee` (figuras 13.19–13.20) deriva da classe `Employee` (linha 8 da Figura 13.19). As implementações de função-membro (Figura 13.20) incluem um construtor (linhas 9–15) que aceita um nome, um sobrenome, um número de seguro social, uma quantidade de vendas e uma taxa de comissão; as funções `set` (linhas 18–21 e 30–33), para atribuir novos valores aos membros de dados `commissionRate` e `grossSales`, respectivamente; as funções `get` (linhas 24–27 e 36–39), que recuperam os valores desses membros de dados; a função `earnings` (linhas 43–46) para calcular os rendimentos de um `CommissionEmployee`; e a função `print` (linhas 49–55), que gera saída do tipo do empregado, a saber, "commission employee: " e informações específicas do empregado. O construtor `CommissionEmployee` também passa o nome, o sobrenome e o número de seguro social para o construtor `Employee` (linha 11) a fim de inicializar os membros de dados `private` do `Employee`. A função `print` chama a função `print` da classe básica (linha 52) para exibir informações específicas do `Employee` (isto é, o nome, o sobrenome e o número de seguro social).

```

1 // Figura 13.17: HourlyEmployee.h
2 // Definição da classe HourlyEmployee.
3 #ifndef HOURLY_H
4 #define HOURLY_H
5
6 #include "Employee.h" // definição da classe Employee
7
8 class HourlyEmployee : public Employee
9 {
10 public:
11     HourlyEmployee( const string &, const string &,
12                    const string &, double = 0.0, double = 0.0 );
13
14     void setWage( double ); // configura o salário por hora
15     double getWage() const; // retorna o salário por hora
16
17     void setHours( double ); // configura as horas trabalhadas
18     double getHours() const; // retorna as horas trabalhadas
19
20     // a palavra-chave virtual assinala intenção de sobrescrever
21     virtual double earnings() const; // calcula os rendimentos
22     virtual void print() const; // imprime o objeto HourlyEmployee
23 private:
24     double wage; // salário por hora
25     double hours; // horas trabalhadas durante a semana
26 }; // fim da classe HourlyEmployee
27
28 #endif // HOURLY_H

```

Figura 13.17 Arquivo de cabeçalho da classe HourlyEmployee.

```

1 // Figura 13.18: HourlyEmployee.cpp
2 // Definições de função-membro da classe HourlyEmployee.
3 #include <iostream>
4 using std::cout;
5
6 #include "HourlyEmployee.h" // definição da classe HourlyEmployee
7
8 // construtor
9 HourlyEmployee::HourlyEmployee( const string &first, const string &last,
10                                const string &ssn, double hourlyWage, double hoursWorked )
11     : Employee( first, last, ssn )
12 {
13     setWage( hourlyWage ); // valida a remuneração por hora
14     setHours( hoursWorked ); // valida as horas trabalhadas
15 } // fim do construtor HourlyEmployee
16
17 // configura a remuneração
18 void HourlyEmployee::setWage( double hourlyWage )
19 {
20     wage = ( hourlyWage < 0.0 ? 0.0 : hourlyWage );
21 } // fim da função setWage
22
23 // retorna a remuneração

```

Figura 13.18 Arquivo de implementação da classe HourlyEmployee.

(continua)

```

24 double HourlyEmployee::getWage() const
25 {
26     return wage;
27 } // fim da função getWage
28
29 // configura as horas trabalhadas
30 void HourlyEmployee::setHours( double hoursWorked )
31 {
32     hours = ( ( hoursWorked >= 0.0 ) && ( hoursWorked <= 168.0 ) ) ?
33         hoursWorked : 0.0 ;
34 } // fim da função setHours
35
36 // retorna as horas trabalhadas
37 double HourlyEmployee::getHours() const
38 {
39     return hours;
40 } // fim da função getHours
41
42 // calcula os rendimentos;
43 // sobrescreve a função virtual pura earnings em Employee
44 double HourlyEmployee::earnings() const
45 {
46     if ( getHours() <= 40 ) // nenhuma hora extra
47         return getWage() * getHours();
48     else
49         return 40 * getWage() + ( ( getHours() - 40 ) * getWage() * 1.5 );
50 } // fim da função earnings
51
52 // imprime informações do HourlyEmployee
53 void HourlyEmployee::print() const
54 {
55     cout << "hourly employee: ";
56     Employee::print(); // reutilização de código
57     cout << "\nhourly wage: " << getWage() <<
58         " "; hours worked: " << getHours();
59 } // fim da função print

```

Figura 13.18 Arquivo de implementação da classe HourlyEmployee.

(continuação)

```

1 // Figura 13.19: CommissionEmployee.h
2 // Classe CommissionEmployee derivada de Employee.
3 #ifndef COMMISSION_H
4 #define COMMISSION_H
5
6 #include "Employee.h" // definição da classe Employee
7
8 class CommissionEmployee : public Employee
9 {
10 public:
11     CommissionEmployee( const string &, const string &,
12         const string &, double = 0.0, double = 0.0 );
13
14     void setCommissionRate( double ); // configura a taxa de comissão
15     double getCommissionRate() const; // retorna a taxa de comissão

```

Figura 13.19 Arquivo de cabeçalho da classe CommissionEmployee.

(continua)

```

16
17     void setGrossSales( double ); // configura a quantidade de vendas brutas
18     double getGrossSales() const; // retorna a quantidade de vendas brutas
19
20     // a palavra-chave virtual assinala intenção de sobrescrever
21     virtual double earnings() const; // calcula os rendimentos
22     virtual void print() const; // imprime o objeto CommissionEmployee
23 private:
24     double grossSales; // vendas brutas semanais
25     double commissionRate; // porcentagem da comissão
26 }; // fim da classe CommissionEmployee
27
28 #endif // COMMISSION_H

```

Figura 13.19 Arquivo de cabeçalho da classe CommissionEmployee.

(continuação)

```

1 // Figura 13.20: CommissionEmployee.cpp
2 // Definições de função-membro da classe CommissionEmployee.
3 #include <iostream>
4 using std::cout;
5
6 #include "CommissionEmployee.h" // definição da classe CommissionEmployee
7
8 // construtor
9 CommissionEmployee::CommissionEmployee( const string &first,
10     const string &last, const string &ssn, double sales, double rate )
11     : Employee( first, last, ssn )
12 {
13     setGrossSales( sales );
14     setCommissionRate( rate );
15 } // fim do construtor CommissionEmployee
16
17 // configura a taxa de comissão
18 void CommissionEmployee::setCommissionRate( double rate )
19 {
20     commissionRate = ( ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0 );
21 } // fim da função setCommissionRate
22
23 // retorna a taxa de comissão
24 double CommissionEmployee::getCommissionRate() const
25 {
26     return commissionRate;
27 } // fim da função getCommissionRate
28
29 // configura a quantidade de vendas brutas
30 void CommissionEmployee::setGrossSales( double sales )
31 {
32     grossSales = ( ( sales < 0.0 ) ? 0.0 : sales );
33 } // fim da função setGrossSales
34
35 // retorna a quantidade de vendas brutas
36 double CommissionEmployee::getGrossSales() const
37 {
38     return grossSales;

```

Figura 13.20 Arquivo de implementação da classe CommissionEmployee.

(continua)

```

39 } // fim da função getGrossSales
40
41 // calcula os rendimentos;
42 // sobrescreve a função virtual pura earnings em Employee
43 double CommissionEmployee::earnings() const
44 {
45     return getCommissionRate() * getGrossSales();
46 } // fim da função earnings
47
48 // imprime informações do CommissionEmployee
49 void CommissionEmployee::print() const
50 {
51     cout << "commission employee: ";
52     Employee::print(); // reutilização de código
53     cout << "\ngross sales: " << getGrossSales()
54         << "; commission rate: " << getCommissionRate();
55 } // fim da função print

```

Figura 13.20 Arquivo de implementação da classe CommissionEmployee.

(continuação)

13.6.5 Criando a classe derivada concreta indireta BasePlusCommissionEmployee

A classe `BasePlusCommissionEmployee` (figuras 13.21–13.22) herda diretamente da classe `CommissionEmployee` (linha 8 da Figura 13.21) e, portanto, é uma classe derivada indireta da classe `Employee`. As implementações de função-membro da classe `BasePlusCommissionEmployee` incluem um construtor (linhas 10–16 da Figura 13.22) que aceita como argumentos um nome, um sobrenome, um número de seguro social, uma quantidade de vendas, uma taxa de comissão e um salário-base. Em seguida, ele passa o nome, o sobrenome, o número de seguro social, a quantidade de vendas e a taxa de comissão para o construtor `CommissionEmployee` (linha 13) para inicializar os membros herdados. `BasePlusCommissionEmployee` também contém uma função `set` (linhas 19–22) para atribuir um novo valor ao membro de dados `baseSalary` e uma função `get` (linhas 25–28) para retornar o valor `baseSalary`. A função `earnings` (linhas 32–35) calcula os rendimentos de um `BasePlusCommissionEmployee`. Observe que a linha 34 na função `earnings` chama a função `earnings` da classe básica `CommissionEmployee` para calcular a parte baseada na comissão dos rendimentos do empregado. Esse é um exemplo elegante de reutilização de código. A função `print` de `BasePlusCommissionEmployee` (linhas 38–43) gera saída de "base-salaried", seguida pela saída da função `print` de classe básica `CommissionEmployee` (outro exemplo de reutilização de código) e, então, o salário-base. A saída resultante começa com "base-salaried commission employee: " seguida pelas demais informações de `BasePlusCommissionEmployee`. Lembre-se de que `print` de `CommissionEmployee` exibe o nome, o sobrenome e o número de seguro social do empregado invocando a função `print` de sua classe básica (isto é, `Employee`) — ainda outro exemplo de reutilização de código. Observe que a função `print` de `BasePlusCommissionEmployee` inicia uma cadeia de chamadas de função que se distribuem por todos os três níveis da hierarquia `Employee`.

13.6.6 Demonstrando o processamento polimórfico

Para testar nossa hierarquia `Employee`, o programa na Figura 13.23 cria um objeto de cada uma das quatro classes concretas `SalariedEmployee`, `HourlyEmployee`, `CommissionEmployee` e `BasePlusCommissionEmployee`. O programa manipula esses objetos, primeiro com a vinculação estática e, então, polimorficamente, utilizando um vector de ponteiros `Employee`. As linhas 31–38 criam objetos de cada uma das quatro classes `Employee` derivadas concretas. As linhas 43–51 geram saída de informações e rendimentos de cada `Employee`. Cada invocação de função-membro nas linhas 43–51 é um exemplo de vinculação estática — em tempo de compilação, porque estamos utilizando handles de nome (não ponteiros ou referências que poderiam ser configurados em tempo de execução), o compilador pode identificar o tipo de cada objeto para determinar que funções `print` e `earnings` são chamadas.

A linha 54 aloca vector `employees`, que contém quatro ponteiros para `Employee`. A linha 57 aponta `employees[ 0 ]` para o objeto `salariedEmployee`. A linha 58 aponta `employees[ 1 ]` para o objeto `hourlyEmployee`. A linha 59 aponta `employees[ 2 ]` para o objeto `commissionEmployee`. A linha 60 aponta `employee[ 3 ]` para o objeto `basePlusCommissionEmployee`. O compilador permite essas atribuições, porque um `SalariedEmployee` é um `Employee`, um `HourlyEmployee` é um `Employee`, um `CommissionEmployee` é um `Employee` e um `BasePlusCommissionEmployee` é um `Employee`. Portanto, podemos atribuir os endereços dos objetos `SalariedEmployee`, `HourlyEmployee`, `CommissionEmployee` e `BasePlusCommissionEmployee` aos ponteiros da classe básica `Employee` (ainda que `Employee` seja uma classe abstrata).

A instrução `for` nas linhas 68–69 percorre vector `employees` e invoca a função `virtualViaPointer` (linhas 83–87) para cada elemento em `employees`. A função `virtualViaPointer` recebe no parâmetro `baseClassPtr` (do tipo `const Employee * const`) o endereço armazenado em um elemento `employees`. Cada chamada a `virtualViaPointer` utiliza `baseClassPtr` para invocar as funções

```

1 // Figura 13.21: BasePlusCommissionEmployee.h
2 // Classe BasePlusCommissionEmployee derivada de Employee.
3 #ifndef BASEPLUS_H
4 #define BASEPLUS_H
5
6 #include "CommissionEmployee.h" // definição da classe CommissionEmployee
7
8 class BasePlusCommissionEmployee : public CommissionEmployee
9 {
10 public:
11     BasePlusCommissionEmployee( const string &, const string &,
12                                const string &, double = 0.0, double = 0.0, double = 0.0 );
13
14     void setBaseSalary( double ); // configura o salário-base
15     double getBaseSalary() const; // retorna o salário-base
16
17     // a palavra-chave virtual assinala intenção de sobrescrever
18     virtual double earnings() const; // calcula os rendimentos
19     virtual void print() const; // imprime o objeto BasePlusCommissionEmployee
20 private:
21     double baseSalary; // salário-base por semana
22 }; // fim da classe BasePlusCommissionEmployee
23
24 #endif // BASEPLUS_H

```

Figura 13.21 Arquivo de cabeçalho da classe BasePlusCommissionEmployee.

```

1 // Figura 13.22: BasePlusCommissionEmployee.cpp
2 // Definições de função-membro BasePlusCommissionEmployee.
3 #include <iostream>
4 using std::cout;
5
6 // definição da classe BasePlusCommissionEmployee
7 #include "BasePlusCommissionEmployee.h"
8
9 // construtor
10 BasePlusCommissionEmployee::BasePlusCommissionEmployee(
11     const string &first, const string &last, const string &ssn,
12     double sales, double rate, double salary )
13     : CommissionEmployee( first, last, ssn, sales, rate )
14 {
15     setBaseSalary( salary ); // valida e armazena o salário-base
16 } // fim do construtor BasePlusCommissionEmployee
17
18 // configura o salário-base
19 void BasePlusCommissionEmployee::setBaseSalary( double salary )
20 {
21     baseSalary = ( ( salary < 0.0 ) ? 0.0 : salary );
22 } // fim da função setBaseSalary
23
24 // retorna o salário-base
25 double BasePlusCommissionEmployee::getBaseSalary() const
26 {
27     return baseSalary;

```

Figura 13.22 Arquivo de implementação da classe BasePlusCommissionEmployee.

(continua)


```

28 } // fim da função getBaseSalary
29
30 // calcula os rendimentos;
31 // sobreescreve a função virtual pura earnings em Employee
32 double BasePlusCommissionEmployee::earnings() const
33 {
34     return getBaseSalary() + CommissionEmployee::earnings();
35 } // fim da função earnings
36
37 // imprime informações de BasePlusCommissionEmployee
38 void BasePlusCommissionEmployee::print() const
39 {
40     cout << "base-salaried ";
41     CommissionEmployee::print(); // reutilização de código
42     cout << "; base salary: " << getBaseSalary();
43 } // fim da função print

```

Figura 13.22 Arquivo de implementação da classe BasePlusCommissionEmployee.

(continuação)

```

1 // Figura 13.23: fig13_23.cpp
2 // Processando objetos da classe derivada Employee individualmente
3 // e polimorficamente utilizando vinculação dinâmica.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7 using std::fixed;
8
9 #include <iomanip>
10 using std::setprecision;
11
12 #include <vector>
13 using std::vector;
14
15 // inclui definições de classes na hierarquia Employee
16 #include "Employee.h"
17 #include "SalariedEmployee.h"
18 #include "HourlyEmployee.h"
19 #include "CommissionEmployee.h"
20 #include "BasePlusCommissionEmployee.h"
21
22 void virtualViaPointer( const Employee * const ); // protótipo
23 void virtualViaReference( const Employee & ); // protótipo
24
25 int main()
26 {
27     // configura a formatação de saída de ponto flutuante
28     cout << fixed << setprecision( 2 );
29
30     // cria objetos da classe derivada
31     SalariedEmployee salariedEmployee(
32         "John", "Smith", "111-11-1111", 800 );
33     HourlyEmployee hourlyEmployee(
34         "Karen", "Price", "222-22-2222", 16.75, 40 );
35     CommissionEmployee commissionEmployee(

```

Figura 13.23 Programa de driver da hierarquia de classes Employee.

(continua)

```

36     "Sue", "Jones", "333-33-3333", 10000, .06 );
37     BasePlusCommissionEmployee basePlusCommissionEmployee(
38         "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
39
40     cout << "Employees processed individually using static binding:\n\n";
41
42     // gera saída de informações e rendimentos dos Employees com vinculação estática
43     salariedEmployee.print();
44     cout << "\nearned $" << salariedEmployee.earnings() << "\n\n";
45     hourlyEmployee.print();
46     cout << "\nearned $" << hourlyEmployee.earnings() << "\n\n";
47     commissionEmployee.print();
48     cout << "\nearned $" << commissionEmployee.earnings() << "\n\n";
49     basePlusCommissionEmployee.print();
50     cout << "\nearned $" << basePlusCommissionEmployee.earnings()
51         << "\n\n";
52
53     // cria um vector a partir dos quatro ponteiros da classe básica
54     vector < Employee * > employees( 4 );
55
56     // inicializa o vector com Employees
57     employees[ 0 ] = &salariedEmployee;
58     employees[ 1 ] = &hourlyEmployee;
59     employees[ 2 ] = &commissionEmployee;
60     employees[ 3 ] = &basePlusCommissionEmployee;
61
62     cout << "Employees processed polymorphically via dynamic binding:\n\n";
63
64     // chama virtualViaPointer para imprimir informações e rendimentos
65     // de cada Employee utilizando vinculação dinâmica
66     cout << "Virtual function calls made off base-class pointers:\n\n";
67
68     for ( size_t i = 0; i < employees.size(); i++ )
69         virtualViaPointer( employees[ i ] );
70
71     // chama virtualViaReference para imprimir informações
72     // de cada Employee utilizando vinculação dinâmica
73     cout << "Virtual function calls made off base-class references:\n\n";
74
75     for ( size_t i = 0; i < employees.size(); i++ )
76         virtualViaReference( *employees[ i ] ); // observe o desreferenciamento
77
78     return 0;
79 } // fim de main
80
81 // chama funções print e earnings virtual de Employee a partir de um
82 // ponteiro de classe básica utilizando vinculação dinâmica
83 void virtualViaPointer( const Employee * const baseClassPtr )
84 {
85     baseClassPtr->print();
86     cout << "\nearned $" << baseClassPtr->earnings() << "\n\n";
87 } // fim da função virtualViaPointer
88
89 // chama funções print e earnings virtual de Employee a partir de uma
90 // referência de classe básica utilizando vinculação dinâmica
91 void virtualViaReference( const Employee &baseClassRef )

```

Figura 13.23 Programa de driver da hierarquia de classes Employee.

(continua)

```

92 {
93     baseClassRef.print();
94     cout << "\nearned $" << baseClassRef.earnings() << "\n\n";
95 } // fim da função virtualViaReference

```

Employees processed individually using static binding:

```

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

```

```

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00
earned $670.00

```

```

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

```

```

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
earned $500.00

```

Employees processed polymorphically using dynamic binding:

Virtual function calls made off base-class pointers:

```

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

```

```

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00
earned $670.00

```

```

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

```

```

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
earned $500.00

```

Virtual function calls made off base-class references:

```

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00

```

```

earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00
earned $670.00
commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
earned $500.00

```

Figura 13.23 Programa de driver da hierarquia de classes Employee.

(continuação)

virtual print (linha 85) e earnings (linha 86). Observe que a função `virtualViaPointer` não contém informações de tipo de `SalariedEmployee`, `HourlyEmployee`, `CommissionEmployee` ou `BasePlusCommissionEmployee`. A função conhece apenas o tipo de classe básica `Employee`. Portanto, em tempo de compilação, o compilador não pode saber que funções da classe concreta chamar por meio de `baseClassPtr`. Ainda em tempo de execução, cada invocação de função virtual chama a função no objeto para o qual `baseClassPtr` aponta nesse momento. A saída ilustra que as funções apropriadas para cada classe são, de fato, invocadas e que informações adequadas de cada objeto são exibidas. Por exemplo, o salário semanal é exibido para o `SalariedEmployee` e as vendas brutas são exibidas para o `CommissionEmployee` e `BasePlusCommissionEmployee`. Observe também que obter os rendimentos de cada `Employee` polimorficamente na linha 86 produz os mesmos resultados que obter os rendimentos desses empregados via vinculação estática nas linhas 44, 46, 48 e 50. Todas as chamadas de função virtual para `print` e `earnings` são convertidas em tempo de execução com vinculação dinâmica.

Por fim, outra instrução `for` (linhas 75–76) percorre `employees` e invoca a função `virtualViaReference` (linhas 91–95) para cada elemento no vector. A função `virtualViaReference` recebe em seu parâmetro `baseClassRef` (de tipo `const Employee &`) uma referência formada desreferenciando o ponteiro armazenado em cada elemento `employees` (linha 76). Cada chamada para `virtualViaReference` invoca as funções `virtual print` (linha 93) e `earnings` (linha 94) via a referência `baseClassRef` para demonstrar que o processamento polimórfico também ocorre com referências de classe básica. Cada invocação de função virtual chama a função no objeto que `baseClassRef` referencia em tempo de execução. Esse é outro exemplo de vinculação dinâmica. A saída produzida utilizando referências de classe básica é idêntica à produzida utilizando ponteiros de classe básica.

13.7 Polimorfismo, funções virtual e vinculação dinâmica ‘sob o capô’ (opcional)

O C++ torna o polimorfismo fácil de programar. É certamente possível programar para polimorfismo em linguagens não orientadas a objeto como C, mas fazer isso requer manipulações de ponteiro complexas e potencialmente perigosas. Esta seção discute como o C++ pode implementar o polimorfismo, as funções virtual e a vinculação dinâmica internamente. Isso fornecerá um entendimento sólido de como essas capacidades realmente funcionam. Mais importante, ajudará você a avaliar o overhead do polimorfismo — em termos de consumo adicional de memória e tempo de processador. Isso ajudará a determinar quando utilizar e quando evitar o polimorfismo. Como você verá no Capítulo 23, “Standard Template Library (STL)”, os componentes STL foram implementados sem polimorfismo e funções virtual — isto foi feito para evitar o overhead de tempo de execução associado e alcançar um ótimo desempenho para atender aos requisitos únicos do STL.

Primeiro, explicaremos as estruturas de dados que o compilador C++ constrói em tempo de compilação para suportar polimorfismo em tempo de execução. Você verá que o polimorfismo é realizado por três níveis de ponteiros (isto é, ‘tripla indireção’). Então mostraremos como um programa em execução utiliza essas estruturas de dados para executar funções virtual e alcançar a vinculação dinâmica associada com polimorfismo. Observe que nossa discussão explica uma possível implementação; esse não é um requisito da linguagem.

Quando o C++ compila uma classe que tem uma ou mais funções virtual, ele constrói uma **tabela de função virtual (vtable)** para essa classe. Um programa em execução utiliza o `vtable` para selecionar a implementação de função adequada toda vez que uma função virtual dessa classe é chamada. A coluna mais à esquerda da Figura 13.24 ilustra o `vtables` para as classes `Employee`, `SalariedEmployee`, `HourlyEmployee`, `CommissionEmployee` e `BasePlusCommissionEmployee`.

Na `vtable` da classe `Employee`, o primeiro ponteiro de função é configurado como 0 (isto é, o ponteiro nulo). Isso é feito porque a função `earnings` é uma função virtual pura e, portanto, não possui uma implementação. O segundo ponteiro de função aponta para a

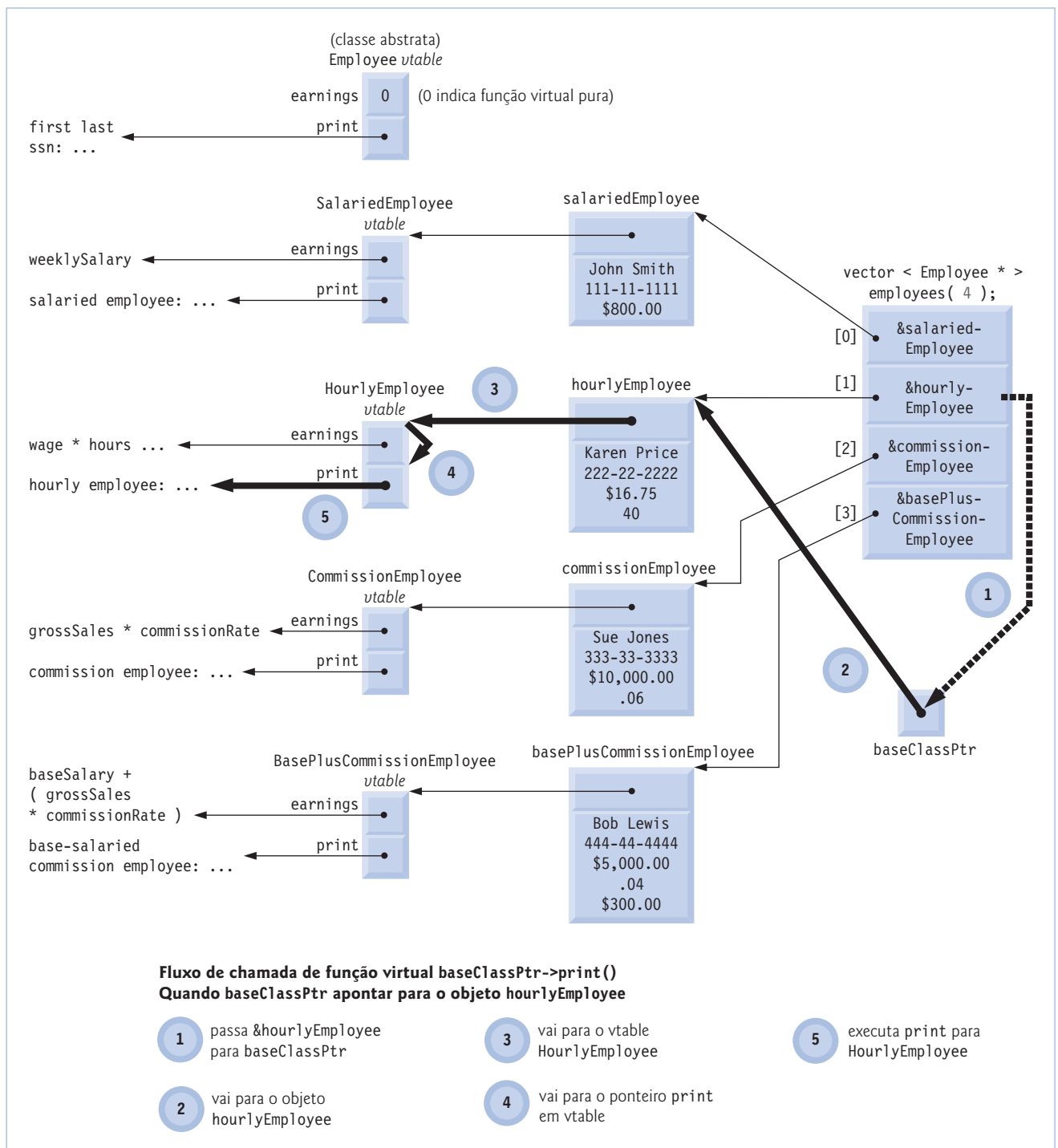


Figura 13.24 Como as chamadas de função virtual funcionam.

função print, que exibe o nome completo e o número de seguro social do empregado. [Nota: Abreviamos a saída de cada função print nessa figura para conservar espaço.] Qualquer classe que tem um ou mais ponteiros nulos em sua vtable é uma classe abstrata. As classes sem ponteiros vtable nulos (como SalariedEmployee, HourlyEmployee, CommissionEmployee e BasePlusCommissionEmployee) são classes concretas.

A classe SalariedEmployee sobrescreve a função earnings para retornar o salário semanal do empregado, assim o ponteiro de função aponta para a função earnings da classe SalariedEmployee. SalariedEmployee também sobrescreve print, então o ponteiro da função correspondente aponta para a função-membro SalariedEmployee que imprime "salaried employee:" seguido pelo nome, número de seguro social e salário semanal do empregado.

O ponteiro da função `earnings` na *vtable* para a classe `HourlyEmployee` aponta para a função `earnings` de `HourlyEmployee` que retorna o `wage` do empregado multiplicado pelo número de `hours` trabalhadas. Observe que, para conservar o espaço, omitimos o fato de que os empregados por hora recebem 50% a mais pelas horas extras trabalhadas. O ponteiro de função `print` aponta para a versão `HourlyEmployee` da função, que imprime "hourly employee: ", o nome, o número de seguro social, o salário por hora e as horas trabalhadas do empregado. Ambas as funções sobrescrevem as funções na classe `Employee`.

O ponteiro de função `earnings` na *vtable* para a classe `CommissionEmployee` aponta para a função `earnings` de `CommissionEmployee` que retorna as vendas brutas do empregado multiplicadas pela taxa de comissão. O ponteiro da função `print` aponta para a versão `CommissionEmployee` da função, que imprime o tipo, o nome, o número de seguro social, a taxa de comissão e as vendas brutas do empregado. Como na classe `HourlyEmployee`, ambas as funções sobrescrevem as funções na classe `Employee`.

O ponteiro da função `earnings` na *vtable* para a classe `BasePlusCommissionEmployee` aponta para a função `earnings` de `BasePlusCommissionEmployee` que retorna as vendas brutas do empregado mais o salário-base multiplicado pela taxa de comissão. O ponteiro da função `print` aponta para a versão `BasePlusCommissionEmployee` da função, que imprime o salário-base mais o tipo, o nome, o número de seguro social, a taxa de comissão e as vendas brutas do empregado. Ambas as funções sobrescrevem as funções na classe `CommissionEmployee`.

Note que, em nosso estudo de caso `Employee`, cada classe concreta fornece sua própria implementação para as funções virtual `earnings` e `print`. Você já aprendeu que cada classe que herda diretamente de classe básica abstrata `Employee` deve implementar `earnings` a fim de ser uma classe concreta, porque `earnings` é uma função virtual pura. Entretanto, essas classes não precisam implementar a função `print` para serem consideradas concretas — `print` não é uma função virtual pura e as classes derivadas podem herdar a implementação de `print` da classe `Employee`. Além disso, a classe `BasePlusCommissionEmployee` não tem de implementar a função `print` ou `earnings` — ambas as implementações de função podem ser herdadas da classe `CommissionEmployee`. Se uma classe em nossa hierarquia tivesse de herdar implementações de função dessa maneira, os ponteiros *vtable* para essas funções simplesmente apontariam para a implementação de função que estava sendo herdada. Por exemplo, se `BasePlusCommissionEmployee` não sobrescrevesse `earnings`, o ponteiro da função `earnings` na *vtable* para a classe `BasePlusCommissionEmployee` apontaria para a mesma função `earnings` para a qual a *vtable* da classe `CommissionEmployee` aponta.

O polimorfismo é realizado por uma estrutura de dados elegante envolvendo três níveis de ponteiros. Discutimos um nível — os ponteiros de função na *vtable*. Esses apontam para as funções reais que executam quando uma função virtual é invocada.

Agora consideremos o segundo nível de ponteiros. Sempre que um objeto de uma classe com uma ou mais funções virtual for instanciado, o compilador anexa ao objeto um ponteiro para a *vtable* dessa classe. Esse ponteiro está normalmente na frente do objeto, mas não é necessário que ele seja implementado dessa maneira. Na Figura 13.24, esses ponteiros são associados com os objetos criados na Figura 13.23 (um objeto para cada um dos tipos `SalariedEmployee`, `HourlyEmployee`, `CommissionEmployee` e `BasePlusCommissionEmployee`). Note que o diagrama exibe cada um dos valores de membro de dados do objeto. Por exemplo, o objeto `salariedEmployee` contém um ponteiro para a *vtable* `SalariedEmployee`; o objeto também contém os valores John Smith, 111-11-1111 e \$800.00.

O terceiro nível de ponteiros simplesmente contém os handles para os objetos que recebem as chamadas de função virtual. Os handles nesse nível também podem ser referências. Observe que a Figura 13.24 retrata o vector `employees` que contém os ponteiros `Employee`.

Agora vejamos como uma típica chamada de função virtual executa. Considere a chamada `baseClassPtr->print()` na função `virtualViaPointer` (linha 85 da Figura 13.23). Suponha que `baseClassPtr` contém `employees[ 1 ]` (isto é, o endereço do objeto `hourlyEmployee` em `employees`). Quando o compilador compila essa instrução, ele determina que a chamada está, de fato, sendo feita por meio de um ponteiro de classe básica e que `print` é uma função virtual.

O compilador determina que `print` é a *segunda* entrada em cada uma das *vtables*. Para localizar essa entrada, o compilador nota que precisará pular a primeira entrada. Portanto, o compilador compila um *offset* ou *deslocamento* de quatro bytes (quatro bytes para cada ponteiro nas máquinas de 32 bits populares atuais, e apenas um ponteiro precisa ser pulado) na tabela de ponteiros de código-objeto de linguagem de máquina para localizar o código que executará a chamada de função virtual.

O compilador gera código que realiza as seguintes operações [Nota: Os números na lista correspondem aos números circulados na Figura 13.24]:

1. Selecione a *i*-ésima entrada de `employees` (nesse caso, o endereço de objeto `hourlyEmployee`) e passe-a como argumento para a função `virtualViaPointer`. Isso configura o parâmetro `baseClassPtr` para apontar para `hourlyEmployee`.
2. Desreferencie esse ponteiro para chegar ao objeto `hourlyEmployee` — que, como você se lembra, começa com um ponteiro para a *vtable* `HourlyEmployee`.
3. Desreferencie o ponteiro *vtable* de `hourlyEmployee` para chegar à *vtable* `HourlyEmployee`.
4. Pule o deslocamento de quatro bytes para selecionar o ponteiro de função `print`.
5. Desreferencie o ponteiro de função `print` para formar o ‘nome’ da função real a executar, e utilize o operador de chamada de função `()` para executar a função `print` apropriada, que nesse caso imprime o tipo, o nome, o número de seguro social, o salário-hora e as horas trabalhadas do empregado.

As estruturas de dados da Figura 13.24 podem parecer complexas, mas essa complexidade é gerenciada pelo compilador e ocultada de você, tornando a programação polimórfica simples e direta. As operações de desreferenciamento de ponteiro e acessos de memória

que ocorrem em cada chamada de função `virtual` exigem um tempo de execução adicional. Os ponteiros `vtables` e `vtable` adicionados aos objetos requerem alguma memória adicional. Agora você tem informações suficientes para determinar se as funções `virtual` são apropriadas aos seus programas.



Dica de desempenho 13.1

O polimorfismo, quando normalmente implementado com funções `virtual` e vinculação dinâmica em C++, é eficiente. Os programadores podem utilizar essas capacidades com impacto nominal no desempenho.



Dica de desempenho 13.2

As funções virtuais e a vinculação dinâmica permitem a programação polimórfica como uma alternativa à programação lógica `switch`. Otimizar compiladores normalmente gera código polimórfico que executa tão bem quanto a lógica baseada em `switch` codificada manualmente. O overhead do polimorfismo é aceitável na maioria dos aplicativos. Mas em algumas situações — aplicativos de tempo real com rigorosos requisitos de desempenho, por exemplo — o overhead de polimorfismo pode ser muito alto.



Observação de engenharia de software 13.11

A vinculação dinâmica permite que os fornecedores de softwares independentes (ISV) distribuam software sem revelar segredos ‘proprietários’ (patenteados). As distribuições de software podem consistir apenas em arquivos de cabeçalho e arquivos de objeto — nenhum código-fonte precisa ser revelado. Os desenvolvedores de software podem então utilizar a herança para derivar novas classes desses fornecidas pelos ISV. Outros softwares que funcionavam com as classes que os ISV forneciam ainda funcionarão com as classes derivadas e utilizarão as funções `virtual` sobrescritas fornecidas nessas classes (via vinculação dinâmica).

13.8 Estudo de caso: sistema de folha de pagamento utilizando polimorfismo e informações de tipo de tempo de execução com `downcasting`, `dynamic_cast`, `typeid` e `type_info`

Considerando a declaração do problema no começo da Seção 13.6, lembre-se de que, para o período atual de salário, nossa empresa fictícia decidiu recompensar `BasePlusCommissionEmployees` adicionando 10% a seus salários-base. Ao processar os objetos `Employee` polimorficamente na Seção 13.6.6, não precisamos nos preocupar com as ‘especificidades’. Agora, porém, para ajustar os salários-base de `BasePlusCommissionEmployees`, temos de determinar o tipo específico de cada objeto `Employee` em tempo de execução e, então, agir apropriadamente. Esta seção demonstra as poderosas capacidades das informações de tipo em tempo de execução (RTTI) e da coerção dinâmica, que permitem a um programa determinar o tipo de um objeto em tempo de execução e atuar nesse objeto de maneira correspondente.

Alguns compiladores, como o Microsoft Visual C++ .NET, requerem que a RTTI seja ativada antes que possa ser utilizada em um programa. Consulte a documentação do compilador para determinar se ele tem requisitos semelhantes. Para ativar a RTTI em Visual C++ .NET, selecione o menu `Project` e, então, selecione a opção de propriedades para o projeto atual. Na caixa de diálogo `Property Pages` que aparece, selecione `Configuration Properties > C/C++ > Language`. Então escolha `Yes (/GR)` a partir da caixa de combinação próximo a `Enable Run-Time Type Info`. Por fim, clique em `OK` para salvar as configurações.

O programa na Figura 13.25 usa a hierarquia `Employee` desenvolvida na Seção 13.6 e aumenta 10% do salário-base de cada `BasePlusCommissionEmployee`. A linha 31 declara vector `employees` de quatro elementos que armazena os ponteiros para o objeto `Employee`. As linhas 34–41 preenchem o vector com os endereços de objetos dinamicamente alocados de classes `SalariesEmployee` (figuras 13.15–13.16), `HourlyEmployee` (figuras 13.17–13.18), `CommissionEmployee` (figuras 13.19–13.20) e `BasePlusCommissionEmployee` (figuras 13.21–13.22).

A instrução `for` nas linhas 44–66 itera pelo `employees` vector e exibe informações de cada `Employee` invocando a função-membro `print` (linha 46). Lembre-se de que, como `print` é declarado `virtual` na classe básica `Employee`, o sistema invoca a função `print` do objeto de classe derivada apropriado.

Neste exemplo, ao encontrarmos os objetos `BasePlusCommissionEmployee`, desejamos aumentar 10% seu salário-base. Visto que processamos os empregados genericamente (isto é, polimorficamente), não podemos (com as técnicas que aprendemos) estar certos do tipo de `Employee` que está sendo manipulado em um dado momento qualquer. Isso cria um problema, porque os empregados `BasePlusCommissionEmployee` devem ser identificados quando os encontramos para que possam receber o aumento de 10% de salário. Para realizar isso, utilizamos o operador `dynamic_cast` (linha 51) a fim de determinar se o tipo de cada objeto é ou não `BasePlusCommissionEmployee`. Essa é a operação `downcast` à qual nos referimos na Seção 13.3.3. As linhas 50–52 fazem `downcast` dinamicamente no `employees[i]` do tipo `Employee *` para o tipo `BasePlusCommissionEmployee *`. Se o elemento vector aponta para um objeto que é um objeto `BasePlusCommissionEmployee`, então o endereço desse objeto é atribuído a `commissionPtr`; caso contrário, 0 é atribuído ao ponteiro de classe derivada `derivedPtr`.


```

1 // Figura 13.25: fig13_25.cpp
2 // Demonstrando downcasting e o RTTI.
3 // NOTA: Para esse exemplo executar em Visual C++ .NET,
4 // você precisa ativar o RTTI (Run-Time Type Info) para o projeto.
5 #include <iostream>
6 using std::cout;
7 using std::endl;
8 using std::fixed;
9
10 #include <iomanip>
11 using std::setprecision;
12
13 #include <vector>
14 using std::vector;
15
16 #include <typeinfo>
17
18 // inclui definições de classes na hierarquia Employee
19 #include "Employee.h"
20 #include "SalariedEmployee.h"
21 #include "HourlyEmployee.h"
22 #include "CommissionEmployee.h"
23 #include "BasePlusCommissionEmployee.h"
24
25 int main()
26 {
27     // configura a formatação de saída de ponto flutuante
28     cout << fixed << setprecision( 2 );
29
30     // cria um vector a partir dos quatro ponteiros da classe básica
31     vector < Employee * > employees( 4 );
32
33     // inicializa vector com vários tipos de Employees
34     employees[ 0 ] = new SalariedEmployee(
35         "John", "Smith", "111-11-1111", 800 );
36     employees[ 1 ] = new HourlyEmployee(
37         "Karen", "Price", "222-22-2222", 16.75, 40 );
38     employees[ 2 ] = new CommissionEmployee(
39         "Sue", "Jones", "333-33-3333", 10000, .06 );
40     employees[ 3 ] = new BasePlusCommissionEmployee(
41         "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
42
43     // processa polimorficamente cada elemento no vector employees
44     for ( size_t i = 0; i < employees.size(); i++ )
45     {
46         employees[ i ]->print(); // gera saída de informações do empregado
47         cout << endl;
48
49         // ponteiro downcast
50         BasePlusCommissionEmployee *derivedPtr =
51             dynamic_cast < BasePlusCommissionEmployee * >
52                 ( employees[ i ] );
53
54         // determina se o elemento aponta para o empregado comissionado com
55         // salário-base
56         if ( derivedPtr != 0 ) // 0 se não for um BasePlusCommissionEmployee

```

Figura 13.25 Demonstrando downcasting e informações de tipo em tempo de execução.

(continua)

```

57     {
58         double oldBaseSalary = derivedPtr->getBaseSalary();
59         cout << "old base salary: $" << oldBaseSalary << endl;
60         derivedPtr->setBaseSalary( 1.10 * oldBaseSalary );
61         cout << "new base salary with 10% increase is: $"
62             << derivedPtr->getBaseSalary() << endl;
63     } // fim do if
64
65     cout << "earned $" << employees[ i ]->earnings() << "\n\n";
66 } // fim do for
67
68 // libera objetos apontados pelos elementos do vector
69 for ( size_t j = 0; j < employees.size(); j++ )
70 {
71     // gera saída do nome de classe
72     cout << "deleting object of "
73         << typeid( *employees[ j ] ).name() << endl;
74
75     delete employees[ j ];
76 } // fim do for
77
78 return 0;
79 } // fim de main

```

```

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: 800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: 16.75; hours worked: 40.00
earned $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: 10000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: 5000.00; commission rate: 0.04; base salary: 300.00
old base salary: $300.00
new base salary with 10% increase is: $330.00
earned $530.00

deleting object of class SalariedEmployee
deleting object of class HourlyEmployee
deleting object of class CommissionEmployee
deleting object of class BasePlusCommissionEmployee

```

Figura 13.25 Demonstrando downcasting e informações de tipo em tempo de execução.

(continuação)

Se o valor retornado pelo operador `dynamic_cast` nas linhas 50–52 não é 0, o objeto é o tipo correto e a instrução `if` (linhas 56–63) realiza o processamento especial requerido para o objeto `BasePlusCommissionEmployee`. As linhas 58, 60 e 62 invocam as funções `BasePlusCommissionEmployee::getBaseSalary` e `BasePlusCommissionEmployee::setBaseSalary` para recuperar e atualizar o salário do empregado.

A linha 65 invoca a função-membro `earnings` no objeto para o qual `employees[ i ]` aponta. Lembre-se de que `earnings` é declarada `virtual` na classe básica, então o programa invoca a função `earnings` do objeto de classe derivada — outro exemplo de vinculação dinâmica.

O loop `for` nas linhas 69–76 exibe o tipo de objeto de cada empregado e utiliza o operador `delete` para desalocar a memória dinâmica para a qual cada elemento `vector` aponta. O operador `typeid` (linha 73) retorna uma referência a um objeto da classe `type_info` que contém as informações sobre o tipo de seu operando, incluindo o nome desse tipo. Quando invocada, a função-membro `type_info::name` (linha 73) retorna uma string baseada em ponteiro que contém o nome de tipo (por exemplo, `"class BasePlusCommissionEmployee"`) do argumento passado para `typeid`. [Nota: O conteúdo exato da string retornada pela função-membro `type_info::name` pode variar de um compilador para o outro.] Para utilizar `typeid`, o programa deve incluir o arquivo de cabeçalho `<typeinfo>` (linha 16).

Observe que evitamos vários erros de compilação neste exemplo fazendo `downcast` de um ponteiro `Employee` para um ponteiro `BasePlusCommissionEmployee` (linhas 50–52). Se removermos o `dynamic_cast` da linha 51 e tentarmos atribuir o ponteiro `Employee` atual diretamente ao ponteiro `BasePlusCommissionEmployee commissionPtr`, receberemos um erro de compilação. O C++ não permite que um programa atribua um ponteiro de classe básica a um ponteiro de classe derivada porque o relacionamento *é um* não se aplica nesse caso — um `CommissionEmployee` *não* é um `BasePlusCommissionEmployee`. Só é possível aplicar o relacionamento *é um* entre a classe derivada e suas classes básicas, não vice-versa.

De maneira semelhante, se as linhas 58, 60 e 62 utilizassem o ponteiro de classe básica atual de `employees`, em vez do ponteiro de classe derivada `commissionPtr`, para invocar as funções `getBaseSalary` e `setBaseSalary` exclusivas de classe derivada, receberíamos um erro de compilação em cada uma dessas linhas. Como você aprendeu na Seção 13.3.3, não é permitido tentar invocar funções exclusivas de classe derivada por meio de um ponteiro de classe básica. Embora as linhas 58, 60 e 62 sejam executadas somente se `commissionPtr` não for 0 (isto é, se for possível realizar a coerção), não podemos tentar invocar as funções de classe derivada `BasePlusCommissionEmployee::getBaseSalary` e `BasePlusCommissionEmployee::setBaseSalary` no ponteiro de classe básica `Employee`. Lembre-se de que, utilizando um ponteiro de classe básica `Employee`, podemos invocar somente as funções localizadas na classe básica `Employee` — as funções `earnings`, `print` e `get` e `set` do `Employee`.

13.9 Destrutores virtuais

Pode ocorrer um problema ao utilizar o polimorfismo para processar objetos dinamicamente alocados de uma hierarquia de classes. Até agora você viu os **destrutores não virtuais** — destrutores que não são declarados com palavra-chave `virtual`. Se um objeto de classe derivada com um destrutor não virtual é destruído explicitamente aplicando o operador `delete` a um ponteiro de classe básica para o objeto, o padrão C++ especifica que o comportamento é indefinido.

Uma solução simples para esse problema é criar um **destrutor virtual** (isto é, um destrutor que é declarado com a palavra-chave `virtual`) na classe básica. Isso torna `virtual` todos os destrutores de classe derivada *mesmo que eles não tenham o mesmo nome do destrutor de classe básica*. Agora, se um objeto na hierarquia é destruído explicitamente aplicando o operador `delete` a um ponteiro de classe básica, o destrutor da classe apropriada é chamado com base no objeto para o qual o ponteiro de classe básica aponta. Lembre-se, quando um objeto de classe derivada é destruído, a parte de classe básica do objeto de classe derivada também é destruída. Desse modo, é importante que tanto os destrutores de classe derivada como os de classe básica executem. O destrutor de classe básica automaticamente executa depois do destrutor de classe derivada.



Boa prática de programação 13.2

Se uma classe tiver funções `virtual`, forneça um destrutor `virtual`, mesmo que ele não seja requerido para a classe. As classes derivadas dessa classe podem conter destrutores que devem ser chamados adequadamente.



Erro comum de programação 13.5

Os construtores não podem ser `virtual`. Declarar um construtor `virtual` é um erro de compilação.

13.10 Estudo de caso de engenharia de software: incorporando herança ao sistema ATM (opcional)

Agora, revisitaremos nosso projeto do sistema ATM para verificar como ele se beneficia da herança. Para aplicar a herança, primeiro procuramos aspectos comuns entre as classes no sistema. Criamos uma hierarquia de herança para modelar classes semelhantes (embora não idênticas) de uma maneira mais eficiente e elegante que nos permite processar os objetos dessas classes polimorficamente. Então modificamos nosso diagrama de classes para incorporar os novos relacionamentos de herança. Por fim, demonstramos como o nosso projeto atualizado é convertido em arquivos de cabeçalho C++.

Na Seção 3.11, encontramos o problema da representação de uma transação financeira no sistema. Em vez de criar uma classe para representar todos os tipos de transação, decidimos criar três classes individuais de transação — `BalanceInquiry`, `Withdrawal` e `Deposit` — para representar as transações que o sistema ATM pode realizar. A Figura 13.26 mostra os atributos e operações dessas classes. Note que elas têm um atributo (`accountNumber`) e uma operação (`execute`) em comum. Toda classe requer o atributo `accountNumber`.

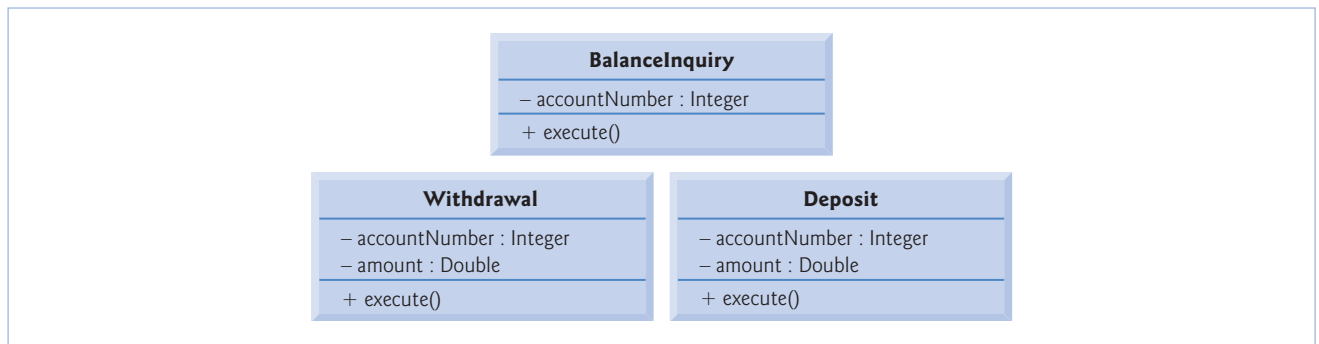


Figura 13.26 Atributos e operações das classes BalanceInquiry, Withdrawal e Deposit.

para especificar a conta à qual a transação se aplica. Cada classe contém uma operação `execute`, que o ATM invoca para realizar uma transação. Claramente, `BalanceInquiry`, `Withdrawal` e `Deposit` representam *tipos de transações*. A Figura 13.26 revela os aspectos comuns entre as classes de transação, portanto utilizar a herança para fatorar os recursos comuns parece apropriado para projetar essas classes. Colocamos as funcionalidades comuns em classe básica `Transaction` e derivamos as classes `BalanceInquiry`, `Withdrawal` e `Deposit` de `Transaction` (Figura 13.27).

A UML especifica um relacionamento chamado **generalização** para modelar a herança. A Figura 13.27 é o diagrama de classes que modela o relacionamento de herança entre classe básica `Transaction` e suas três classes derivadas. As setas com pontas vazadas triangulares indicam que as classes `BalanceInquiry`, `Withdrawal` e `Deposit` são derivadas da classe `Transaction`. Diz-se que a classe `Transaction` deve ser uma generalização de suas classes derivadas. Diz-se que as classes derivadas são **especializações** da classe `Transaction`.

As classes `BalanceInquiry`, `Withdrawal` e `Deposit` compartilham o atributo do tipo inteiro `accountNumber`, assim dividimos esse atributo comum e o colocamos na classe de base `Transaction`. Não listamos mais `accountNumber` no segundo compartimento de cada classe derivada, porque as três classes derivadas herdam este atributo de `Transaction`. Lembre-se, porém, de que as classes derivadas não podem acessar o atributo `private` de uma classe básica. Portanto, incluímos a função-membro `public getAccountNumber` na classe `Transaction`. Toda classe derivada herda essa função-membro, permitindo que a classe derivada acesse seu `accountNumber` quando for necessário executar uma transação.

De acordo com a Figura 13.26, as classes `BalanceInquiry`, `Withdrawal` e `Deposit` também compartilham a operação `execute`, então a classe básica `Transaction` deve conter a função-membro `public execute`. Entretanto, não faz sentido implementar `execute` na classe `Transaction`, porque a funcionalidade que essa função-membro fornece depende do tipo específico da transação real. Portanto, declaramos a função-membro `execute` como uma função virtual pura na classe básica `Transaction`. Isso torna `Transaction` uma classe abstrata e força qualquer classe derivada de `Transaction`, que deve ser uma classe concreta (isto é, `BalanceInquiry`, `Withdrawal` e `Deposit`), a implementar a função-membro virtual `execute` pura para tornar concreta a classe derivada. A UML requer que coloquemos em *itálico* os nomes de classe abstrata (e funções virtual puras — **operações abstratas** na UML). Desse modo, `Transaction` e sua função-membro `execute` aparecem em *itálico* na Figura 13.27. Observe que a operação `execute` não é escrita em *itálico* na classe derivada `BalanceInquiry`, `Withdrawal` e `Deposit`. Cada classe derivada sobrescreve a função-membro `execute` da classe básica `Transaction` com uma implementação apropriada. Observe que a Figura 13.27 inclui a operação `execute` no terceiro

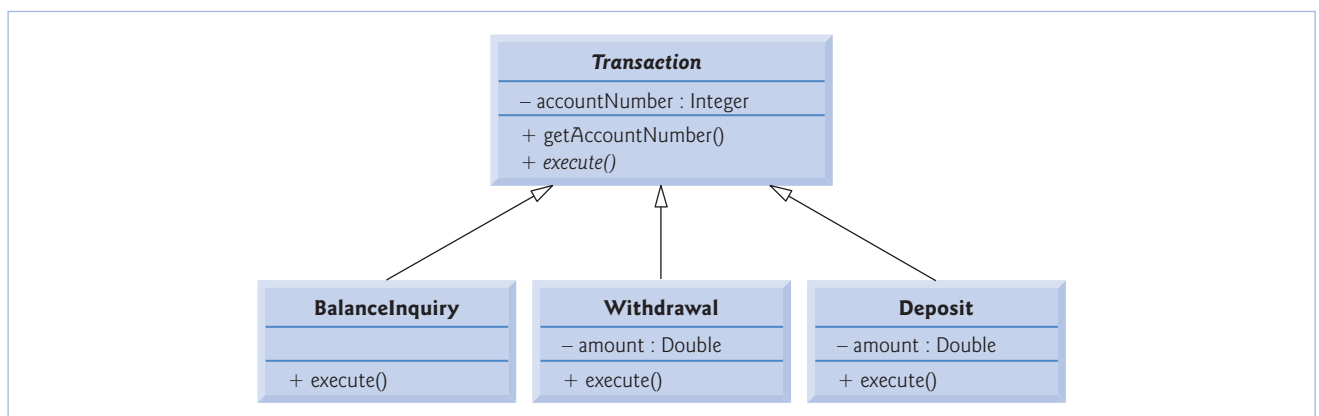


Figura 13.27 O diagrama de classes modela o relacionamento de generalização entre a classe básica `Transaction` e as classes derivadas `BalanceInquiry`, `Withdrawal` e `Deposit`.

compartimento das classes `BalanceInquiry`, `Withdrawal` e `Deposit`, pois cada classe tem uma implementação concreta diferente da função-membro sobrescrita.

Como você aprendeu neste capítulo, uma classe derivada pode herdar a interface ou a implementação de uma classe básica. Comparada com uma hierarquia projetada para herança de implementação, aquela projetada para herança de interface tende a ter suas funcionalidades na parte inferior da hierarquia — uma classe básica significa que uma ou mais funções devem ser definidas por classes na hierarquia, mas as classes derivadas individuais fornecem suas próprias implementações da(s) função(ões). A hierarquia de herança projetada para o sistema ATM tira proveito desse tipo de herança, que fornece ao ATM uma maneira elegante de executar todas as transações ‘de modo geral’. Cada classe derivada de `Transaction` herda alguns detalhes de implementação (por exemplo, o membro de dados `accountNumber`), mas o principal benefício de incorporar herança em nosso sistema é que as classes derivadas compartilham uma interface comum (por exemplo, a função-membro `virtual execute` pura). O ATM pode apontar um ponteiro `Transaction` em qualquer transação e, quando o ATM invoca `execute` por esse ponteiro, a versão de `execute` apropriada a essa transação (isto é, implementada no arquivo `.cpp` dessa classe derivada) executa automaticamente. Por exemplo, suponha que um usuário opte por realizar uma consulta de saldos. O ATM aponta um ponteiro `Transaction` para um novo objeto da classe `BalanceInquiry`, que o compilador C++ permite porque um `BalanceInquiry` é uma `Transaction`. Quando o ATM utiliza esse ponteiro para invocar `execute`, a versão de `execute` de `BalanceInquiry` é chamada.

Essa abordagem polimórfica também torna o sistema facilmente extensível. Se quiséssemos criar um novo tipo de transação (por exemplo, uma transferência de fundos ou um pagamento de conta), simplesmente criaríamos uma classe derivada `Transaction` adicional que sobrescrevesse a função-membro `execute` com uma versão apropriada ao novo tipo de transação. Só seriam necessárias alterações mínimas no código do sistema para permitir que os usuários escolham o novo tipo de transação no menu principal e para que o ATM instancie e execute objetos da nova classe derivada. O ATM poderia executar transações do novo tipo utilizando o código atual, porque ele executa todas as transações de modo idêntico.

Como você aprendeu anteriormente neste capítulo, uma classe abstrata como `Transaction` é uma em que o programador nunca desejará instanciar objetos. Uma classe abstrata simplesmente declara atributos e comportamentos comuns para suas classes derivadas em uma hierarquia de herança. A classe `Transaction` define o conceito de uma transação com um número de conta e a executa. Você pode se perguntar por que nos incomodamos em incluir a função-membro `virtual execute` pura na classe `Transaction` se `execute` não possui uma implementação concreta. Conceitualmente, incluímos essa função-membro porque ela é o comportamento definidor de todas as transações — em execução. Tecnicamente, devemos incluir a função-membro `execute` na classe básica `Transaction` para que o ATM (ou qualquer outra classe) possa invocar polimorficamente a versão sobrescrita de cada classe derivada dessa função por uma referência ou ponteiro `Transaction`.

As classes derivadas `BalanceInquiry`, `Withdrawal` e `Deposit` herdam o atributo `accountNumber` da classe básica `Transaction`, mas as classes `Withdrawal` e `Deposit` contêm o atributo adicional `amount` que as distingue da classe `BalanceInquiry`. As classes `Withdrawal` e `Deposit` requerem esse atributo adicional para armazenar a quantia que o usuário deseja sacar ou depositar. A classe `BalanceInquiry` não precisa desse atributo e requer somente um número de conta para executar. Apesar de duas das três classes derivadas `Transaction` compartilharem esse atributo, não a colocamos na classe básica `Transaction` — colocamos apenas os recursos comuns a *todas* as classes derivadas na classe básica. Desse modo, as classes derivadas não herdam atributos (e operações) desnecessários.

A Figura 13.28 apresenta um diagrama de classes atualizado do nosso modelo que incorpora a herança e introduz a classe `Transaction`. Modelamos uma associação entre a classe ATM e a classe `Transaction` para mostrar que a ATM, em um dado momento, executa ou não uma transação (isto é, há zero ou um objeto do tipo `Transaction` no sistema por vez). Como um `Withdrawal` é um tipo de `Transaction`, não desenhamos mais uma linha de associação diretamente entre a classe ATM e a classe `Withdrawal` — a classe derivada `Withdrawal` herda a associação da classe básica `Transaction` com a classe ATM. As classes derivadas `BalanceInquiry` e `Deposit` também herdam essa associação, que substitui as associações anteriormente omitidas entre as classes `BalanceInquiry` e `Deposit` e a classe ATM. Observe novamente o uso de setas de pontas triangulares vazadas para indicar as especializações da classe `Transaction`, como indicado na Figura 13.27.

Também adicionamos uma associação entre as classes `Transaction` e `BankDatabase` (Figura 13.28). Todas as classes `Transaction` exigem uma referência a `BankDatabase` de modo que possam acessar e modificar as informações de conta. Cada classe `Transaction` derivada herda essa referência, então não modelamos mais a associação entre a classe `Withdrawal` e a `BankDatabase`. Observe que a associação entre a classe `Transaction` e a `BankDatabase` substitui as associações anteriormente omitidas entre as classes `BalanceInquiry` e `Deposit` e a `BankDatabase`.

Incluímos uma associação entre a classe `Transaction` e `Screen` porque todas as `Transactions` exibem saída para o usuário por meio de `Screen`. Cada classe derivada herda essa associação. Portanto, não incluímos mais a associação anteriormente modelada entre `Withdrawal` e `Screen`. Entretanto, a classe `Withdrawal` ainda participa das associações com `CashDispenser` e `Keypad` — essas associações se aplicam à classe derivada `Withdrawal`, mas não às classes derivadas `BalanceInquiry` e `Deposit`, por isso não movemos essas associações para a classe básica `Transaction`.

Nosso diagrama de classes que incorpora herança (Figura 13.28) também modela `Deposit` e `BalanceInquiry`. Mostramos as associações entre `Deposit`, `DepositSlot` e `Keypad`. Observe que a classe `BalanceInquiry` não aceita nenhuma associação diferente daquelas herdadas da classe `Transaction` — um `BalanceInquiry` interage somente com `BankDatabase` e `Screen`.

O diagrama de classes da Figura 9.20 mostrou atributos e operações com marcadores de visibilidade. Agora apresentamos um diagrama de classes modificado na Figura 13.29 que inclui a classe básica abstrata `Transaction`. Esse diagrama abreviado não mostra relacionamentos de herança (esses aparecem na Figura 13.28), mas em vez disso mostra os atributos e operações depois que empregamos

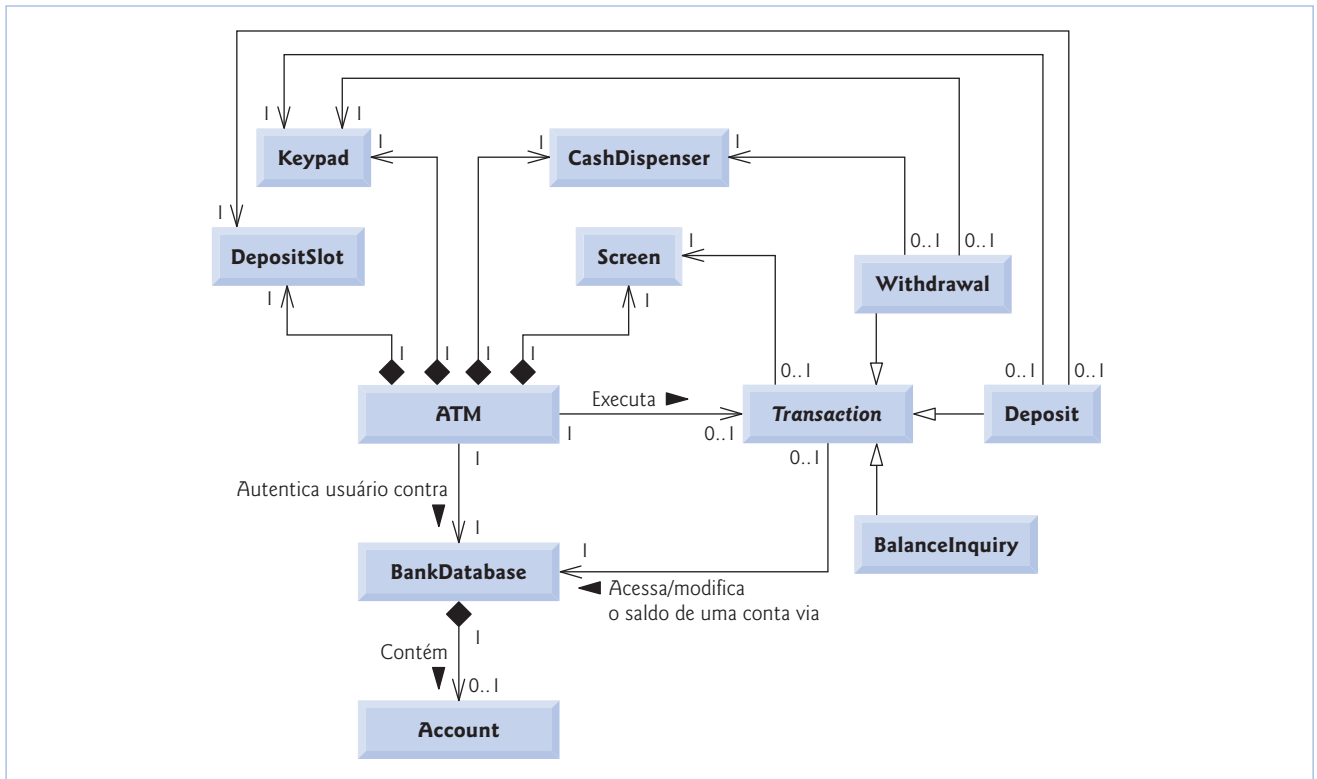


Figura 13.28 O diagrama de classes do sistema ATM (incorporando a herança). Observe que o nome da classe abstrata *Transaction* aparece em itálico.

a herança no sistema. Observe que o nome de classe abstrata *Transaction* e o nome da operação abstrata *execute* na classe *Transaction* aparecem em itálico. Para economizar espaço, como fizemos na Figura 4.24, não incluímos esses atributos mostrados pelas associações na Figura 13.28 — mas os incluímos na implementação C++ do Apêndice G. Omitimos também todos os parâmetros de operação, como fizemos na Figura 9.20 — incorporar herança não afeta os parâmetros já modelados nas figuras 6.22–6.25.



Observação de engenharia de software 13.12

Um diagrama de classes completo mostra todas as associações entre as classes e todos os atributos e operações para cada classe. Se o número de atributos de classe, operações e associações for muito alto (como nas figuras 13.28 e 13.29), uma boa prática que promove a legibilidade é dividir essas informações entre dois diagramas de classes — um focalizando as associações e o outro, os atributos e operações. Entretanto, ao examinar classes modeladas dessa maneira, é crucial considerar ambos os diagramas de classe para ter uma visão completa das classes. Por exemplo, deve-se consultar a Figura 13.28 para observar o relacionamento de herança entre *Transaction* e suas classes derivadas, que é omitido na Figura 13.29.

Implementando o projeto do sistema ATM incorporando a herança

Na Seção 9.12, começamos a implementar o projeto do sistema ATM no código C++. Agora, modificaremos nossa implementação para incorporar a herança, utilizando a classe *Withdrawal* como um exemplo.

1. Se uma classe A é uma generalização da classe B, então a classe B é derivada da classe A (e é uma especialização dela). Por exemplo, a classe básica abstrata *Transaction* é uma generalização da classe *Withdrawal*. Portanto, a classe *Withdrawal* é derivada de (e é uma especialização da) classe *Transaction*. A Figura 13.30 contém uma parte do arquivo de cabeçalho da classe *Withdrawal*, em que a definição de classe indica o relacionamento de herança entre *Withdrawal* e *Transaction* (linha 9).
2. Se a classe A é uma classe abstrata e a classe B é derivada da classe A, então a classe B deve implementar as funções virtuais puras da classe A se a classe B for uma classe concreta. Por exemplo, a classe *Transaction* contém a função virtual *execute* pura, então a classe *Withdrawal* deve implementar essa função-membro se quisermos instanciar um objeto *Withdrawal*. A Figura 13.31 contém o arquivo de cabeçalho C++ da classe *Withdrawal* das figuras 13.28 e 13.29. A classe *Withdrawal* herda o membro de dados *accountNumber* da classe básica *Transaction*, então *Withdrawal* não declara esse membro de dados. A classe *Withdrawal* também herda referências a *Screen* e *BankDatabase* de sua classe básica *Transaction*, então não incluímos

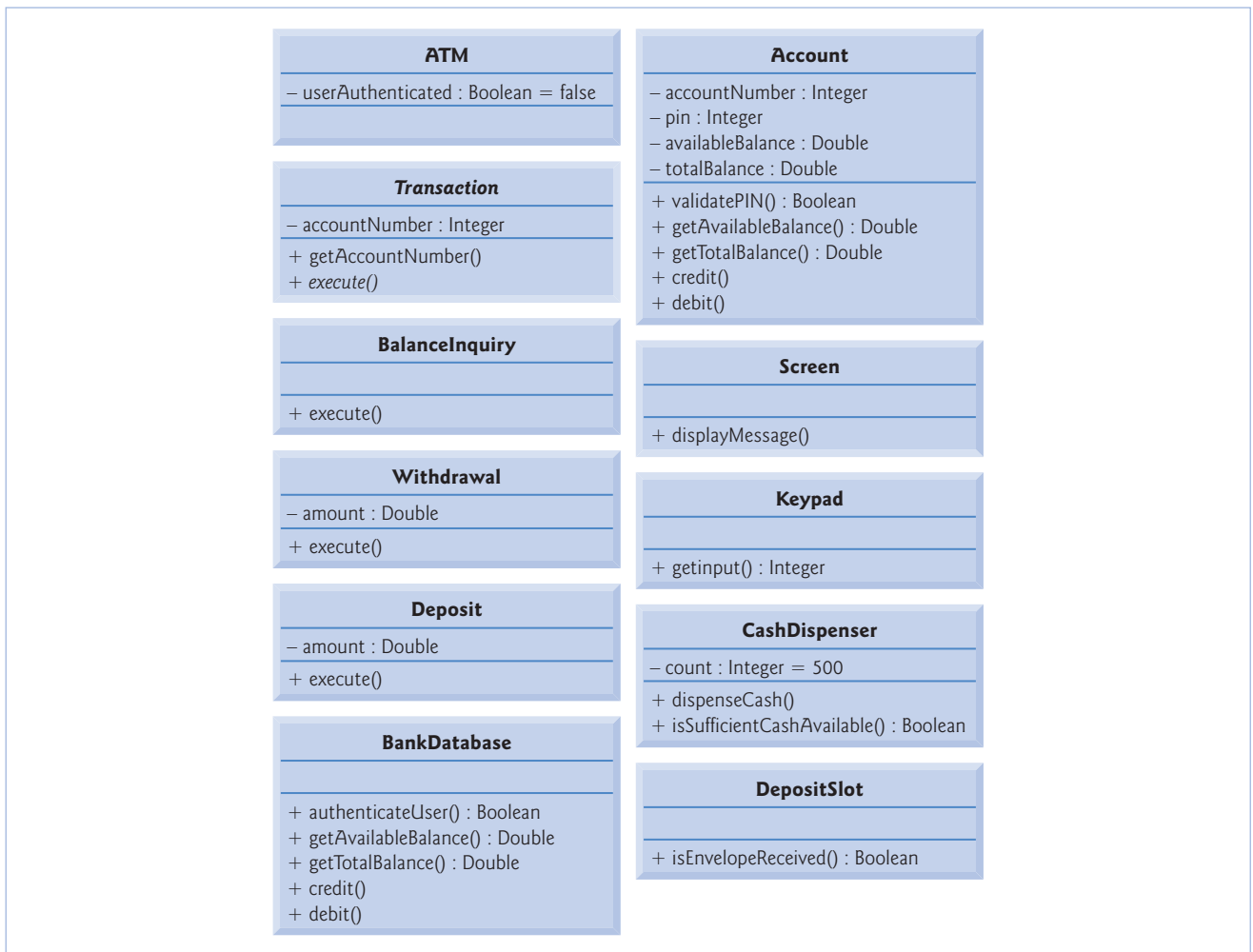


Figura 13.29 Diagrama de classes depois de incorporar herança no sistema.

```

1 // Figura 13.30: Withdrawal.h
2 // Definição da classe Withdrawal que representa uma transação de retirada
3 #ifndef WITHDRAWAL_H
4 #define WITHDRAWAL_H
5
6 #include "Transaction.h" // definição da classe Transaction
7
8 // a classe Withdrawal deriva da classe básica Transaction
9 class Withdrawal : public Transaction
10 {
11 }; // fim da classe Withdrawal
12
13 #endif // WITHDRAWAL_H
  
```

Figura 13.30 Definição da classe Withdrawal que deriva de Transaction.

essas referências no código. A Figura 13.29 especifica o atributo `amount` e a operação `execute` para a classe `Withdrawal`. A linha 19 da Figura 13.31 declara um membro de dados para o atributo `amount`. A linha 16 contém o protótipo de função para a operação `execute`. Lembre-se de que, para ser concreta, uma classe derivada `Withdrawal` deve fornecer uma implementação concreta da função virtual pura `execute` na classe básica `Transaction`. O protótipo na linha 16 sinaliza sua intenção de


```

1 // Figura 13.31: Withdrawal.h
2 // Definição da classe Withdrawal que representa uma transação de retirada
3 #ifndef WITHDRAWAL_H
4 #define WITHDRAWAL_H
5
6 #include "Transaction.h" // definição da classe Transaction
7
8 class Keypad; // declaração antecipada da classe Keypad
9 class CashDispenser; // declaração antecipada da classe CashDispenser
10
11 // a classe Withdrawal deriva da classe básica Transaction
12 class Withdrawal : public Transaction
13 {
14 public:
15     // função-membro que sobrescreve execute na classe básica Transaction
16     virtual void execute(); // realiza a transação
17 private:
18     // atributos
19     double amount; // quantia a sacar
20     Keypad &keypad; // referência ao teclado do ATM
21     CashDispenser &cashDispenser; // referência ao dispensador de notas do ATM
22 }; // fim da classe Withdrawal
23
24 #endif // WITHDRAWAL_H

```

Figura 13.31 O arquivo de cabeçalho da classe `Withdrawal` baseado nas figuras 13.28 e 13.29.

sobrescrever a função `virtual` pura da classe básica. Você deve fornecer esse protótipo se fornecer uma implementação no arquivo `.cpp`. Apresentamos essa implementação no Apêndice G. As referências `keypad` e `cashDispenser` (linhas 20–21) são membros de dados derivados de associações de `Withdrawal` na Figura 13.28. Na implementação dessa classe no Apêndice G, um construtor inicializa essas referências como objetos reais. Mais uma vez, para ser capaz de compilar as declarações das referências nas linhas 20–21, incluímos as declarações dianteiras nas linhas 8–9.

Resumo do estudo de caso ATM

Isso conclui nosso projeto orientado a objetos do sistema ATM. Uma implementação C++ completa do sistema ATM em 877 linhas de código aparece no Apêndice G. Essa implementação funcional utiliza noções-chave de programação, incluindo classes, objetos, encapsulamento, visibilidade, composição, herança e polimorfismo. O código é abundantemente comentado e se adapta às práticas de codificação que você aprendeu. Dominar esse código é uma experiência-chave maravilhosa para você após estudar os capítulos 1–13.

Exercícios de revisão do estudo de caso de engenharia de software

- 13.1** A UML utiliza uma seta com uma _____ para indicar um relacionamento de generalização.
- ponta de seta com preenchimento sólido
 - ponta de seta triangular oca
 - ponta de seta oca na forma de losango
 - ponta de seta angular
- 13.2** Determine se a seguinte sentença é *verdadeira* ou *falsa* e, se *falsa*, explique por quê: A UML exige que os nomes de classe abstrata e os nomes de operação sejam sublinhados.
- 13.3** Escreva um arquivo de cabeçalho C++ para começar a implementar o projeto da classe `Transaction` especificado nas figuras 13.28 e 13.29. Não deixe de incluir as referências `private` baseadas nas associações da classe `Transaction`. Não deixe de incluir também as funções `public get` para todos os membros de dados `private` que as classes derivadas devem acessar para realizar suas tarefas.

Respostas dos exercícios de revisão do estudo de caso de engenharia de software

- 13.1** b.
- 13.2** Falsa. A UML requer que os nomes de classe abstrata e os nomes de operação sejam escritos em itálico.

- 13.3** O projeto para classe `Transaction` produz o arquivo de cabeçalho na Figura 13.32. Na implementação apresentada no Apêndice G, um construtor inicializa os atributos de referência `private screen` e `bankDatabase` como objetos reais e as funções-membro `getScreen` e `getBankDatabase` acessam esses atributos. Essas funções-membro permitem às classes derivadas de `Transaction` acessar a tela do ATM e interagir com o banco de dados do banco.

13.11 Síntese

Neste capítulo discutimos o polimorfismo, que permite ‘programar no geral’ em vez de ‘programar no específico’, e mostramos como isso torna os programas mais extensíveis. Começamos com um exemplo de como o polimorfismo permitiria que um gerenciador de tela exibisse vários objetos ‘espaciais’. Demonstramos então como os ponteiros de classes básicas e derivadas podem ser apontados para objetos de classes básicas e derivadas. Dissemos que é natural apontar ponteiros de classe básica para objetos de classe básica, assim como o é apontar ponteiros de classe derivada para objetos de classe derivada. Apontar ponteiros de classe básica para objetos de classe derivada também é natural porque um objeto de classe derivada *é um* objeto de sua classe básica. Você aprendeu por que é perigoso apontar ponteiros de classe derivada para objetos de classe básica e por que o compilador não autoriza essas atribuições. Introduzimos as funções `virtual`, que permitem que funções adequadas sejam chamadas quando objetos em vários níveis de uma hierarquia de herança são referenciados (em tempo de execução) por meio de ponteiros de classe básica. Isso é conhecido como vinculação dinâmica ou tardia. Em seguida, discutimos as funções `virtual` puras (funções `virtual` que não fornecem uma implementação) e as classes abstratas (classes com uma ou mais funções `virtual` puras). Você aprendeu que as classes abstratas não podem ser utilizadas para instanciar objetos, embora as classes concretas possam. Então demonstramos o uso de classes abstratas em uma hierarquia de herança. Você aprendeu como o polimorfismo funciona ‘sob o capô’ com *vtables* que são criadas pelo compilador. Discutimos como fazer downcasting de ponteiros de classe básica para ponteiros de classe derivada para permitir que um programa chame as funções-membro exclusivas da classe derivada. O capítulo concluiu com uma discussão sobre destrutores `virtual` e como eles asseguram que todos os destrutores apropriados em uma hierarquia de herança executem em um objeto de classe derivada quando esse objeto é excluído via um ponteiro de classe básica.

No próximo capítulo, discutimos os templates, um sofisticado recurso do C++ que permite aos programadores definir uma família de classes relacionadas ou funções com um único segmento de código.

```

1 // Figura 13.32: Transaction.h
2 // Definição da classe básica abstrata Transaction.
3 #ifndef TRANSACTION_H
4 #define TRANSACTION_H
5
6 class Screen; // declaração antecipada da classe Screen
7 class BankDatabase; // declaração antecipada da classe BankDatabase
8
9 class Transaction
10 {
11 public:
12     int getAccountNumber(); // retorna o número da conta
13     Screen &getScreen(); // retorna a referência à tela
14     BankDatabase &getBankDatabase(); // retorna a referência ao banco de dados do banco
15
16     // função virtual pura para realizar a transação
17     virtual void execute() = 0; // sobrescrita em classes derivadas
18 private:
19     int accountNumber; // indica conta envolvida
20     Screen &screen; // referência à tela do ATM
21     BankDatabase &bankDatabase; // referência ao banco de dados de info da conta
22 }; // fim da classe Transaction
23
24 #endif // TRANSACTION_H

```

Figura 13.32 O arquivo de cabeçalho da classe `Transaction` baseado nas figuras 13.28 e 13.29.

Resumo

- Com as funções `virtual` e o polimorfismo, torna-se possível projetar e implementar sistemas que são mais facilmente extensíveis. Os programas podem ser escritos para processar objetos de tipos que podem não existir quando o programa está em desenvolvimento.
- A programação polimórfica com funções `virtual` pode eliminar a necessidade da lógica `switch`. O programador pode utilizar o mecanismo da função `virtual` para realizar a lógica equivalente automaticamente, evitando assim os tipos de erro comumente associados com a lógica `switch`.
- As classes derivadas podem fornecer suas próprias implementações de uma função `virtual` de classe básica se necessário, mas, se não fornecerem, a implementação da classe básica é utilizada.
- Se uma função `virtual` é chamada referenciando um objeto específico por nome e utilizando o operador de seleção de membro ponto, a referência é resolvida em tempo de compilação (isso é chamado de vinculação estática); a função `virtual` que é chamada é aquela definida para a classe desse objeto particular.
- Em muitas situações é útil definir classes abstratas para as quais o programador nunca pretende criar objeto. Como são utilizadas somente como classes básicas, referimo-nos a elas como classes básicas abstratas. Nenhum objeto de uma classe abstrata pode ser instanciado.
- As classes a partir das quais os objetos podem ser instanciados são chamadas classes concretas.
- Uma classe é tornada abstrata declarando uma ou mais de suas funções `virtual` como puras. Uma função `virtual` pura é uma função com um especificador puro (`= 0`) em sua declaração.
- Se uma classe é derivada de uma classe com uma função `virtual` pura e essa classe derivada não fornecer uma definição para essa função `virtual` pura, então essa função `virtual` permanece pura na classe derivada. Consequentemente, a classe derivada também é uma classe abstrata.
- O C++ permite o polimorfismo — a capacidade que objetos de classes diferentes relacionados por herança têm de responder diferentemente à mesma chamada de função-membro.
- O polimorfismo é implementado por meio das funções `virtual` e da vinculação dinâmica.
- Quando uma solicitação é feita por uma referência ou ponteiro de classe básica para utilizar uma função `virtual`, o C++ escolhe a função sobrescrita correta na classe derivada apropriada associada com o objeto.
- Pelo uso de funções `virtual` e do polimorfismo, uma chamada de função-membro pode produzir ações diferentes, dependendo do tipo do objeto que recebe a chamada.
- Embora não possamos instanciar objetos de classes básicas abstratas, podemos declarar ponteiros e referências para objetos de classes básicas abstratas. Esses ponteiros e referências podem ser utilizados para permitir manipulações polimórficas de objetos de classe derivada instanciados a partir de classes derivadas concretas.
- A vinculação dinâmica requer que, em tempo de execução, a chamada para uma função `virtual` de membro seja roteada para a versão da função `virtual` apropriada da classe. Uma tabela de função `virtual` chamada *vtable* é implementada como um array que contém ponteiros de função. Toda classe com funções `virtual` tem uma *vtable*. Para cada função `virtual` na classe, a *vtable* tem uma entrada contendo um ponteiro de função para a versão da função `virtual` utilizar para um objeto dessa classe. A função `virtual` a utilizar para uma classe particular poderia ser a função definida nessa classe ou uma função herdada direta ou indiretamente de uma classe básica na parte superior da hierarquia.
- Quando uma classe básica fornece uma função-membro `virtual`, as classes derivadas podem sobrescrever a função `virtual`, mas elas não têm de sobrescrevê-la. Portanto, uma classe derivada pode utilizar a versão de uma classe básica de uma função `virtual`.
- Todo objeto de uma classe com funções `virtual` contém um ponteiro para a *vtable* dessa classe. Quando uma chamada de função é feita de um ponteiro de classe básica para um objeto de classe derivada, o ponteiro de função apropriado na *vtable* é obtido e desreferenciado para completar a chamada em tempo de execução. Essa pesquisa da *vtable* e esse desreferenciamento de ponteiro requerem um overhead nominal de tempo de execução.
- Qualquer classe que tenha um ou mais ponteiros `0` em sua *vtable* é uma classe abstrata. As classes sem ponteiros *vtable* `0` são classes concretas.
- Novos tipos de classes são regularmente adicionados aos sistemas. As novas classes são acomodadas por vinculação dinâmica (também chamada vinculação tardia). O tipo de um objeto precisa ser conhecido em tempo de compilação para uma chamada de função `virtual` ser compilada. Em tempo de execução, a função-membro apropriada será chamada para o objeto para o qual o ponteiro aponta.
- O operador `dynamic_cast` verifica o tipo do objeto para o qual o ponteiro aponta, então, determina se esse tipo tem um relacionamento *é um* com o tipo para o qual o ponteiro está sendo convertido. Se houver um relacionamento *é um*, `dynamic_cast` retorna o endereço do objeto. Se não, `dynamic_cast` retorna `0`.
- O operador `typeid` retorna uma referência a um objeto da classe `type_info` que contém as informações sobre o tipo de seu operando, incluindo o nome do tipo. Para utilizar `typeid`, o programa deve incluir o arquivo de cabeçalho `<typeinfo>`.
- Quando invocada, a função-membro `type_info::name` retorna uma string baseada em ponteiro que contém o nome do tipo que o objeto `type_info` representa.

- Os operadores `dynamic_cast` e `typeid` fazem parte do recurso RTTI (*run-time type information*) do C++ que permite a um programa determinar um tipo de objeto em tempo de execução.
- Declare o destrutor de classe básica `virtual` se a classe contiver funções `virtual`. Isso torna virtuais todos os destrutores de classe derivada, mesmo que eles não tenham o mesmo nome do destrutor da classe básica. Se um objeto na hierarquia é destruído explicitamente aplicando o operador `delete` a um ponteiro de classe básica para um objeto de classe derivada, o destrutor para a classe apropriada é chamado. Depois que um destrutor de classe derivada executa, os destrutores de todas as classes básicas dessa classe executam completamente em ordem ascendente pela hierarquia — o destrutor da classe-raiz executa por último.

Terminologia

<typeinfo>, arquivo de cabeçalho	fluxo de controle de uma chamada de função <code>virtual</code>	programar no geral
classe abstrata	função <code>virtual</code> puraimplementação, herança	RTTI (<i>run-time type information</i>)
classe básica abstrata	interface, herança	sobrescrever uma função
classe concreta	manipulação de ponteiro perigosa	<code>switch</code> , lógica
classe iteradora	name, função, da classe <code>type_info</code>	<code>type_info</code> , classe
coerção dinâmica	operador <code>typeid</code>	vinculação dinâmica
deslocamento	polimorfismo	vinculação estática
deslocamento em uma <code>vtable</code>	polimorfismo como uma alternativa à lógica de <code>switch</code>	vinculação tardia
destrutor não virtual	ponteiro de classe básica para um objeto de classe básica	<code>virtual</code> , destrutor
determinar dinamicamente a função a executar	ponteiro de classe básica para um objeto de classe derivada	<code>virtual</code> , função
downcasting	ponteiro de classe derivada para um objeto de classe básica	<code>virtual</code> , tabela de função (<code>vtable</code>)
<code>dynamic_cast</code>	ponteiro de classe derivada para um objeto de classe derivada	<code>virtual</code> , palavra-chave
especificador puro	ponteiro <code>vtable</code> do objeto	<code>vtable</code>
	programação polimórfica	<code>vtable</code> , ponteiro
	programar no específico	

Exercícios de revisão

- 13.1** Preencha as lacunas em cada uma das seguintes sentenças:
- Tratar um objeto de classe básica como um _____ pode causar erros.
 - O polimorfismo ajuda a eliminar a lógica _____.
 - Se uma classe contém pelo menos uma função `virtual` pura, ela é uma classe _____.
 - As classes a partir das quais os objetos podem ser instanciados são chamadas _____.
 - O operador _____ pode ser utilizado para fazer downcast de ponteiros de classe básica de modo seguro.
 - O operador `typeid` retorna uma referência para um objeto _____.
 - _____ envolve utilizar uma referência ou ponteiro de classe básica para invocar funções `virtual` em objetos de classe básica e de classe derivada.
 - Funções que podem ser sobrescritas são declaradas com a palavra-chave _____.
 - A coerção de um ponteiro de classe básica em um ponteiro de classe derivada é chamada _____.
- 13.2** Determine se cada uma das seguintes sentenças é verdadeira ou falsa. Se falsa, explique por quê.
- Todas as funções `virtual` em uma classe básica abstrata devem ser declaradas como funções `virtual` puras.
 - É perigoso referenciar um objeto de classe derivada com um handle de classe básica.
 - Uma classe torna-se abstrata declarando essa como classe `virtual`.
 - Se uma classe básica declara uma função `virtual` pura, uma classe derivada deve implementar essa função para tornar-se uma classe concreta.
 - A programação polimórfica pode eliminar a necessidade da lógica `switch`.

Respostas dos exercícios de revisão

- 13.1** a) objeto de classe derivada. b) `switch`. c) abstrata. d) concretas. e) `dynamic_cast`. f) `type_info`. g) Polimorfismo. h) `virtual`. i) downcasting.
- 13.2** a) Falsa. Uma classe básica abstrata pode incluir funções virtuais com implementações. b) Falsa. Referenciar um objeto de classe básica com um handle de classe derivada é perigoso. c) Falsa. As classes nunca são declaradas `virtual`. Em vez disso, uma classe é tornada abstrata pela inclusão de pelo menos uma função `virtual` pura na classe. d) Verdadeira. e) Verdadeira.

Exercícios

- 13.3** Como o polimorfismo permite programar ‘no geral’ em vez de ‘no específico?’ Discuta as vantagens-chave da programação ‘no geral’.
- 13.4** Discuta os problemas de programação com a lógica `switch`. Explique por que o polimorfismo pode ser uma alternativa efetiva ao uso da lógica `switch`.
- 13.5** Diferencie herdar interface e herdar implementação. Como as hierarquias de herança projetadas para herdar interface diferem daquelas projetadas para herdar implementação?
- 13.6** O que são funções `virtual`? Descreva uma circunstância em que funções `virtual` seriam apropriadas.
- 13.7** Diferencie vinculação estática e vinculação dinâmica. Explique o uso de funções `virtual` e a `vtable` na vinculação dinâmica.
- 13.8** Diferencie funções `virtual` e funções `virtual` puras.
- 13.9** Sugira um ou mais níveis de classes básicas abstratas para a hierarquia Forma discutida neste capítulo e mostrada na Figura 12.3. (O primeiro nível é Forma e o segundo nível consiste nas classes `FormaBidimensional` e `FormaTridimensional`.)
- 13.10** Como o polimorfismo promove extensibilidade?
- 13.11** Você foi solicitado a desenvolver um simulador de voo que terá saídas gráficas elaboradas. Explique por que a programação polimórfica seria especialmente eficaz para um problema dessa natureza.
- 13.12** (*Modificação de sistema de folha de pagamento*) Modifique o sistema de folha de pagamento das figuras 13.13–13.23 para incluir o membro de dados `private birthDate` na classe `Employee`. Utilize a classe `Date` das figuras 11.12–11.13 para representar o aniversário de um empregado. Suponha que a folha de pagamento seja processada uma vez por mês. Crie um vector de referências `Employee` para armazenar os vários objetos `employee`. Em um loop, calcule a folha de pagamento para cada `Employee` (polimorficamente) e adicione um bônus de \$ 100 ao pagamento do funcionário se o mês atual for o mês em que o ocorre o aniversário do `Employee`.
- 13.13** (*Hierarquia Forma*) Implemente a hierarquia Forma projetada no Exercício 12.7 (que é baseada na hierarquia da Figura 12.3). Cada `FormaBidimensional` deve conter a função `obterArea` para calcular a área da forma bidimensional. Cada `FormaTridimensional` deve ter funções-membro `obterArea` e `obterVolume` para calcular a área do volume e da superfície, respectivamente, da forma tridimensional. Crie um programa que utilize um vector de ponteiros Forma para objetos de cada classe concreta na hierarquia. O programa deve imprimir o objeto para o qual cada elemento vector aponta. Além disso, no loop que processa todas as formas no vector, determine se cada forma é uma `FormaBidimensional` ou `FormaTridimensional`. Se uma forma for uma `FormaBidimensional`, exiba sua área. Se uma forma for uma `FormaTridimensional`, exiba sua área e volume.
- 13.14** (*Gerenciador de tela polimórfico usando a hierarquia Forma*) Desenvolva um pacote básico de imagens gráficas. Use a hierarquia Forma implementada no Exercício 13.13. Limite-se a formas bidimensionais como quadrados, retângulos, triângulos e círculos. Interaja com o usuário. Deixe o usuário especificar a posição, o tamanho, a forma e os caracteres de preenchimentos a serem utilizados no desenho de cada forma. O usuário pode especificar a mesma forma mais de uma vez. Quando criar cada forma, coloque um ponteiro Forma * para cada novo objeto Forma em um array. Cada classe Forma deve ter agora sua própria função-membro `desenhar`. Escreva um gerenciador de tela polimórfico que percorre o array enviando mensagens de `desenhar` para cada objeto no array para formar uma imagem na tela. Redesenhe a imagem da tela toda vez que o usuário especificar uma forma adicional.
- 13.15** (*Hierarquia de herança Package*) Use a hierarquia de herança Package criada no Exercício 12.9 para criar um programa que exibe as informações de endereço e calcula os custos de entrega de vários Packages. O programa deve conter um vector de ponteiros Package para objetos das classes `TwoDayPackage` e `OvernightPackage`. Faça um loop pelo vector para processar o Packages polimorficamente. Para cada Package, invoque as funções `get` para obter as informações de endereço do remetente e do destinatário, e então imprima os dois endereços da maneira que apareceriam nos pacotes de correio. Além disso, chame a função-membro `calculateCost` de cada Package e imprima o resultado. Monitore o custo de entrega total de todos os Packages no vector e exiba esse total quando o loop terminar.
- 13.16** (*Programa polimórfico de operações bancárias usando a hierarquia Account*) Desenvolva um programa polimórfico de operações bancárias usando a hierarquia Account criada no Exercício 12.10. Crie um vector de ponteiros Account para os objetos `SavingsAccount` e `CheckingAccount`. Para cada Account no vector, permita ao usuário especificar um valor em dinheiro a ser retirado de Account utilizando a função-membro `debit` e uma quantia em dinheiro a ser depositada em Account utilizando a função-membro `credit`. Enquanto processa cada Account, determine seu tipo. Se um Account é um `SavingsAccount`, calcule a quantia de juros devida à Account utilizando a função-membro `calculateInterest` e, então, adicione os juros ao saldo da conta utilizando a função-membro `credit`. Depois de processar um Account, imprima o saldo atualizado da conta obtido invocando a função-membro da classe básica `getBalance`.



*Por trás daquele padrão externo
formas indistintas ficam mais
claras a cada dia.*

*É sempre a mesma forma, só
que muito numerosa.*

Charlotte Perkins Gilman

*Todo gênio vê o mundo de um
ângulo diferente de seus pares.*
Havelock Ellis

*... nossa individualidade
especial, enquanto distinta da
nossa humanidade genérica.*
Oliver Wendell Holmes, Sr

Templates

OBJETIVOS

Neste capítulo, você aprenderá:

- A utilizar templates de função para criar convenientemente um grupo de funções relacionadas (sobrecarregadas).
- A distinguir entre templates de função e especializações de template de função.
- A utilizar templates de classe para criar um grupo de tipos relacionados.
- A distinguir entre templates de classe e especializações de template de classe.
- A sobrecarregar templates de função.
- A entender os relacionamentos entre templates, friends, herança e membros estáticos.

- 14.1** Introdução
- 14.2** Templates de funções
- 14.3** Sobrecarregando templates de função
- 14.4** Templates de classe
- 14.5** Parâmetros sem tipo e tipos-padrão para templates de classes
- 14.6** Notas sobre templates e herança
- 14.7** Notas sobre templates e friends
- 14.8** Notas sobre templates e membros `static`
- 14.9** Síntese

Resumo | Terminologia | Exercícios de revisão | Respostas dos exercícios de revisão | Exercícios

14.1 Introdução

Neste capítulo, discutimos um dos mais poderosos recursos de reutilização de software do C++, a saber, os **templates**. Os **templates de função** e os **templates de classe** permitem aos programadores especificar, com um único segmento de código, uma série inteira de funções relacionadas (sobrecarregadas) — chamadas **especializações de template de função** — ou uma série inteira de classes relacionadas — chamadas **especializações de template de classe**. Essa técnica é chamada **programação genérica**.

Poderíamos escrever um único template de função para uma função de classificação de array e, então, fazer o C++ gerar especializações separadas de template de função que classificariam os arrays `int`, `float`, `string` e assim por diante. Introduzimos os templates de função no Capítulo 6. Apresentamos uma discussão e um exemplo adicional neste capítulo.

Poderíamos escrever um único template de classe para uma classe de pilha e, então, fazer o C++ gerar especializações separadas de template de classe, como uma pilha de classe `int`, uma de `float`, uma de `string` e assim por diante.

Observe a distinção entre templates e especializações de template: os templates de função e os templates de classe são como estênceis a partir dos quais traçamos formas; especializações de template de função e especializações de template de classe são como traçados separados que têm a mesma forma, mas poderiam, por exemplo, ser desenhados em cores diferentes.

Neste capítulo, apresentamos um template de função e um template de classe. Consideramos também os relacionamentos entre templates e outros recursos do C++, como sobrecarga, herança, friends e membros `static`. O projeto e os detalhes dos mecanismos de template discutidos aqui são baseados no trabalho de Bjarne Stroustrup, tal como apresentado em seu artigo *Parameterized Types for C++* e publicado nos *Proceedings of the USENIX C++ Conference*, realizada em Denver, Colorado, em outubro de 1988.

Este capítulo é apenas uma introdução a templates. O Capítulo 23, “Standard Template Library (STL)”, apresenta um tratamento aprofundado das classes contêineres de template, iteradores e algoritmos do STL. O Capítulo 23 contém dúzias de exemplos baseados em template de código ativo (*live-code*) que ilustram técnicas mais sofisticadas de programação de template do que as utilizadas aqui.



Observação de engenharia de software 14.1

A maioria dos compiladores C++ requer a exibição da definição completa de um template no arquivo de código-fonte cliente que utiliza o template. Por essa razão e por questão de reusabilidade, os templates são frequentemente definidos em arquivos de cabeçalho, que são então incluídos (com `#include`) nos arquivos de código-fonte cliente apropriados. Para os templates de classe, isso quer dizer que as funções-membro também são definidas no arquivo de cabeçalho.

14.2 Templates de funções

As funções sobrecarregadas normalmente realizam operações *semelhantes* ou *idênticas* sobre tipos de dados diferentes. Se forem *idênticas* para cada tipo, as operações podem ser expressas de maneira mais compacta e conveniente utilizando templates de função. Inicialmente, o programador escreve uma única definição de template de função. Com base nos tipos de argumentos fornecidos explicitamente ou inferidos de chamadas para essa função, o compilador gera funções de código-objeto separadas (isto é, especializações de template de função) para tratar cada chamada de função de modo apropriado. Em C, essa tarefa pode ser realizada utilizando **macros** criadas com a diretiva de pré-processador `#define` (ver o Apêndice F, “Pré-processador”). Entretanto, as macros podem ter efeitos colaterais sérios e, de fato, não permitem ao compilador realizar a verificação de tipos. Os templates de função fornecem uma solução compacta, como as macros, mas permitem verificação de tipos.



Dica de prevenção de erro 14.1

Os templates de função, como as macros, permitem a reutilização de software. Ao contrário das macros, os templates de função ajudam a eliminar muitos tipos de erros pelo minucioso exame de verificação de tipos completa do C++.

Todas as **definições de template de função** começam com a palavra-chave **template** seguida por uma lista de **parâmetros de template** para o template de função entre **colchetes angulares** (< e >); cada parâmetro de template que representa um tipo deve ser precedido por uma das palavras-chave intercambiáveis **class** ou **typename**, como em

```
template< typename T >
```

ou

```
template< class ElementType >
```

ou

```
template< typename BorderType, typename FillType >
```

Os parâmetros de template de tipo de uma definição de template de função são utilizados para especificar os tipos dos argumentos para a função, especificar o tipo de retorno da função e declarar variáveis dentro da função. A definição de função segue e se parece com qualquer outra definição de função. Observe que as palavras-chave **typename** e **class** utilizadas para especificar parâmetros de template de função, na realidade, significam 'qualquer tipo predefinido ou tipo definido pelo usuário'.



Erro comum de programação 14.1

*Não colocar a palavra-chave **class** ou **typename** antes de cada parâmetro de template de tipo de um template de função é um erro de sintaxe.*

Exemplo: template de função **printArray**

Examinemos o template de função **printArray** na Figura 14.1, linhas 8–15. O template de função **printArray** declara (linha 8) um único parâmetro de template **T** (o **T** pode ser qualquer identificador válido) para o tipo do array a ser impresso pela função **printArray**; **T** é referido como um **parâmetro de template de tipo** ou parâmetro de tipo. Você verá os parâmetros de template não-tipo na Seção 14.5.

Ao detectar uma invocação da função **printArray** no programa-cliente (por exemplo, linhas 30, 35 e 40), o compilador utiliza suas capacidades de resolução de sobrecarga para localizar uma definição da função **printArray** que melhor corresponda à chamada de função. Nesse caso, a única função **printArray** com o número apropriado de parâmetros é o template de função **printArray** (linhas 8–15). Considere a chamada de função na linha 30. O compilador compara o tipo do primeiro argumento de **printArray** (**int *** na linha 30) com o primeiro parâmetro do template de função **printArray** (**const T *** na linha 9) e deduz que substituir o parâmetro de tipo **T** pelo **int** faria o argumento corresponder ao parâmetro. Então, o compilador substitui **int** por **T** em toda definição de template e compila uma especialização de **printArray** que pode exibir um array de valores **int**. Na Figura 14.1, o compilador cria três especializações de **printArray** — uma que espera um array **int**, uma que espera um array **double**, e uma que espera um array **char**. Por exemplo, a especialização de template de função para o tipo **int** é

```
void printArray( const int *array, int count )
{
    for ( int i = 0; i < count; i++ )
        cout << array[ i ] << " ";

    cout << endl;
} // fim da função printArray
```

O nome de um parâmetro de template só pode ser declarado uma vez na lista de parâmetros de template de um cabeçalho de template, mas pode ser utilizado repetidamente no cabeçalho e no corpo da função. Os nomes de parâmetros de template entre os templates de função não precisam ser únicos.

A Figura 14.1 demonstra o template de função **printArray** (linhas 8–15). O programa começa declarando o array **int** **a** de cinco elementos, o array **double** **b** de sete elementos e o array **char** **c** de seis elementos (linhas 23–25, respectivamente). Então, o programa gera saída de cada array chamando **printArray** — uma vez com um primeiro argumento **a** do tipo **int *** (linha 30), uma vez com um primeiro argumento **b** do tipo **double *** (linha 35) e uma vez com um primeiro argumento **c** de tipo **char *** (linha 40). A chamada na linha 30, por exemplo, faz com que o compilador infira que **T** é **int** e instancie uma especialização do template de função **printArray**, para a qual o parâmetro de tipo **T** é **int**. A chamada na linha 35 faz com que o compilador infira que **T** é **double** e instancie uma segunda especialização de template da função **printArray**, para a qual o parâmetro de tipo **T** é **double**. A chamada na linha 40 faz com que o compilador infira que **T** é **char** e instancie uma terceira especialização da template de função **printArray**, para a qual o parâmetro de tipo **T** é **char**. É importante observar que, se **T** (linha 8) representa um tipo definido pelo usuário (o que ele não representa na Figura 14.1), deve haver um operador de inserção de fluxo sobrecarregado para esse tipo; caso contrário, o primeiro operador de inserção de fluxo na linha 12 não compilará.



Erro comum de programação 14.2

*Se um template é invocado com um tipo definido pelo usuário e se esse template utilizar funções ou operadores (por exemplo, **==**, **+**, **<=**) com objetos desse tipo de classe, então essas funções e operadores devem ser sobrecarregados para o tipo definido pelo usuário. Esquecer de sobrecarregar esses operadores causa erros de compilação.*

```

1 // Figura 14.1: fig14_01.cpp
2 // Utilizando funções de template.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 // Definição da template de função printArray
8 template< typename T >
9 void printArray( const T *array, int count )
10 {
11     for ( int i = 0; i < count; i++ )
12         cout << array[ i ] << " ";
13
14     cout << endl;
15 } // fim do template de função printArray
16
17 int main()
18 {
19     const int ACOUNT = 5; // tamanho do array a
20     const int BCOUNT = 7; // tamanho do array b
21     const int CCOUNT = 6; // tamanho do array c
22
23     int a[ ACOUNT ] = { 1, 2, 3, 4, 5 };
24     double b[ BCOUNT ] = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
25     char c[ CCOUNT ] = "HELLO"; // posição 6 para null
26
27     cout << "Array a contains:" << endl;
28
29     // chama a especialização do template de função do tipo inteiro
30     printArray( a, ACOUNT );
31
32     cout << "Array b contains:" << endl;
33
34     // chama a especialização do template de função do tipo double
35     printArray( b, BCOUNT );
36
37     cout << "Array c contains:" << endl;
38
39     // chama a especialização do template de função do tipo caractere
40     printArray( c, CCOUNT );
41     return 0;
42 } // fim de main

```

```

Array a contains:
1 2 3 4 5
Array b contains:
1.1 2.2 3.3 4.4 5.5 6.6 7.7
Array c contains:
H E L L O

```

Figura 14.1 Especializações de template de função do template de função printArray.

Neste exemplo, o mecanismo de template poupa o programador de escrever três funções sobrecarregadas separadas com os protótipos

```
void printArray( const int *, int );
void printArray( const double *, int );
void printArray( const char *, int );
```

todos os quais utilizam o mesmo código, exceto pelo tipo T (como utilizado na linha 9).



Dica de desempenho 14.1

Embora os templates ofereçam os benefícios da reusabilidade de software, lembre-se de que múltiplas especializações de template de função e de classe são instanciadas em um programa (em tempo de compilação), apesar de o template ser escrito uma única vez. Essas cópias podem consumir memória considerável. Mas, em geral, isso não é um problema, porque o código gerado pelo template tem o mesmo tamanho do código que o programador teria escrito para produzir as funções sobrecarregadas separadas.

14.3 Sobrecarregando templates de função

Os templates de função e a sobrecarga de função estão intimamente relacionados. Todas as especializações de template de função geradas a partir de um template de função têm o mesmo nome, portanto o compilador utiliza a solução de sobrecarga para invocar a função adequada.

Um template de função pode ser sobrecarregado de várias maneiras. Podemos fornecer outros templates de função que especificam o mesmo nome de função mas parâmetros de função diferentes. Por exemplo, o template de função `printArray` da Figura 14.1 poderia ser sobrecarregado com outro template de função `printArray` com os parâmetros adicionais `lowSubscript` e `highSubscript` para especificar a parte do array a ser gerada para a saída (ver Exercício 14.4).

Um template de função também pode ser sobrecarregado fornecendo funções não-template com o mesmo nome de função, mas com diferentes argumentos de função. Por exemplo, o template de função `printArray` da Figura 14.1 poderia ser sobrecarregado com uma versão não-template que imprime especificamente um array de strings de caracteres em um formato tabular organizado (ver Exercício 14.5).

O compilador realiza um processo de correspondência para determinar que função chamar quando uma função é invocada. Primeiro, o compilador localiza todos os templates de função que correspondam à função identificada na chamada de função e cria especializações baseadas nos argumentos da chamada de função. Então, o compilador localiza todas as funções usuais que correspondem à função identificada na chamada de função. Se uma das funções usuais ou uma das especializações de template de função for a melhor correspondência para a chamada de função, essa função usual ou especialização é utilizada. Se uma função usual e uma especialização forem correspondências igualmente boas para a chamada de função, então a função usual é utilizada. Caso contrário, se houver múltiplas correspondências para a chamada de função, o compilador considera a chamada ambígua e gera uma mensagem de erro.



Erro comum de programação 14.3

Se nenhuma definição de função correspondente puder ser localizada para uma chamada de função particular, ou se houver múltiplas correspondências, o compilador gera um erro.

14.4 Templates de classe

É possível entender o conceito de ‘pilha’ (uma estrutura de dados na qual inserimos itens na parte superior e os recuperamos na ordem último a entrar, primeiro a sair) independentemente do tipo dos itens que são colocados na pilha. Entretanto, para instanciar uma pilha, um tipo de dados deve ser especificado. Isso cria uma maravilhosa oportunidade para a reusabilidade de software. Precisamos de meios para descrever as noções de uma pilha genericamente e instanciar classes que são versões específicas de tipo dessa classe de pilha genérica. O C++ fornece essa capacidade por meio dos templates de classe.



Observação de engenharia de software 14.2

Os templates de classe encorajam a reusabilidade de software permitindo que versões específicas de tipo de classes genéricas sejam instanciadas.

Os templates de classe são chamados de **tipos parametrizados**, porque requerem um ou mais parâmetros de tipo para especificar como personalizar um template de ‘classe genérica’ para formar uma especialização de template de classe.

O programador que quiser produzir uma variedade de especializações de template de classe escreverá somente uma definição de template de classe. Toda vez que uma especialização de template de classe adicional for necessária, o programador utilizará uma notação simples e concisa, e o compilador escreverá o código-fonte para a especialização requerida pelo programador. Um template de classe `Stack`, por exemplo, poderia, então, tornar-se a base para criar muitas classes `Stack` (como ‘`Stack de double`’, ‘`Stack de int`’, ‘`Stack de char`’, ‘`Stack de Employee`’ etc.) utilizadas em um programa.

*Criando o template de classe **Stack**< T >*

Observe a definição do template de classe `Stack` na Figura 14.2. Ela é parecida com uma definição de classe convencional, exceto pelo fato de que é precedida pelo cabeçalho (linha 6)

```
template< typename T >
```

para especificar uma definição de template de classe com o parâmetro de tipo `T` que atua como um marcador de lugar para o tipo da classe `Stack` a ser criada. O programador não precisa utilizar especificamente o identificador `T` — qualquer identificador válido pode ser utilizado. O tipo de elemento a ser armazenado nessa `Stack` é mencionado genericamente como `T` em todo o cabeçalho de classe `Stack` e definições de função-membro. Daqui a pouco, mostraremos como o `T` torna-se associado com um tipo específico, como `double` ou `int`. Devido à maneira que esse template de classe é projetado, há duas restrições para os tipos de dados não fundamentais utilizados com essa `Stack` — eles devem ter um construtor-padrão (para utilizar na linha 44 a fim de criar o array que armazena os elementos de pilha) e devem suportar o operador de atribuição (linhas 56 e 70).

As definições de função-membro de um template de classe são templates de função. As definições de função-membro que aparecem fora da definição de template de classe iniciam, cada uma, com o cabeçalho

```
template< typename T >
```

(linhas 40, 51 e 65). Portanto, cada definição assemelha-se a uma definição de função convencional, exceto pelo fato de que o tipo de elemento `Stack` é sempre listado genericamente como o parâmetro de tipo `T`. O operador binário de resolução de escopo é utilizado com o nome de template de classe `Stack< T >` (linhas 41, 52 e 66) para associar cada definição de função-membro ao escopo do template de classe. Nesse caso, o nome genérico da classe é `Stack< T >`. Quando `doubleStack` é instanciado como tipo `Stack< double >`, a especialização de template de função de construtor `Stack` utiliza `new` para criar um array de elementos de tipo `double` para representar a pilha (linha 44). A instrução

```
stackPtr = new T[ size ];
```

na definição de template de classe `Stack` é gerada pelo compilador na especialização de template de classe `Stack< double >` como

```
stackPtr = new double[ size ];
```

```

1 // Figura 14.2: Stack.h
2 // Template de classe Stack.
3 #ifndef STACK_H
4 #define STACK_H
5
6 template< typename T >
7 class Stack
8 {
9 public:
10     Stack( int = 10 ); // construtor-padrão (tamanho de Stack 10)
11
12     // destrutor
13     ~Stack()
14     {
15         delete [] stackPtr; // desaloca o espaço interno para Stack
16     } // fim do destrutor ~Stack
17
18     bool push( const T& ); // insere (push) um elemento na Stack
19     bool pop( T& ); // remove (pop) um elemento da Stack
20
21     // determina se a Stack está vazia
22     bool isEmpty() const
23     {
24         return top == -1;
25     } // fim da função isEmpty
26
27     // determina se a Stack está cheia
28     bool isFull() const
29     {
```

Figura 14.2 Template de classe `Stack`.

(continua)

```

30     return top == size - 1;
31 } // fim da função isFull
32
33 private:
34     int size; // número de elementos na Stack
35     int top; // localização do elemento superior (-1 significa vazio)
36     T *stackPtr; // ponteiro para a representação interna da Stack
37 }; // fim do template de classe Stack
38
39 // template construtor
40 template< typename T >
41 Stack< T >::Stack( int s )
42     : size( s > 0 ? s : 10 ), // valida o tamanho
43       top( -1 ), // Stack inicialmente vazia
44       stackPtr( new T[ size ] ) // aloca memória para elementos
45 {
46     // corpo vazio
47 } // fim do template construtor Stack
48
49 // insere elemento na Stack;
50 // se bem-sucedida, retorna true; caso contrário, retorna false
51 template< typename T >
52 bool Stack< T >::push( const T &pushValue )
53 {
54     if ( !isFull() )
55     {
56         stackPtr[ ++top ] = pushValue; // insere item em Stack
57         return true; // inserção bem-sucedida
58     } // fim do if
59
60     return false; // inserção malsucedida
61 } // fim do template de função push
62
63 // remove elemento da Stack;
64 // se bem-sucedida, retorna true; caso contrário, retorna false
65 template< typename T >
66 bool Stack< T >::pop( T &popValue )
67 {
68     if ( !isEmpty() )
69     {
70         popValue = stackPtr[ top-- ]; // remove item da Stack
71         return true; // remoção bem-sucedida
72     } // fim do if
73
74     return false; // remoção malsucedida
75 } // fim do template de função pop
76
77 #endif

```

Figura 14.2 Template de classe Stack.

(continuação)

*Criando um driver para testar o template de classe **Stack< T >***

Agora, vamos considerar o driver (Figura 14.3) que executa o template de classe Stack. O driver começa instanciando o objeto `doubleStack` de tamanho 5 (linha 11). Esse objeto é declarado como da classe `Stack< double >` (pronunciado como 'Stack de double'). O compilador associa o tipo `double` com o parâmetro de tipo `T` no template de classe a fim de produzir o código-fonte para uma classe Stack do tipo `double`. Embora os templates ofereçam os benefícios da reusabilidade de software, lembre-se de que múltiplas especializações de

template de classe são instanciadas em um programa (em tempo de compilação), mesmo que o template seja escrito somente uma vez.

As linhas 17–21 invocam push para colocar os valores double 1.1, 2.2, 3.3, 4.4 e 5.5 sobre doubleStack. O loop while termina quando o driver tenta inserir (push) um sexto valor em doubleStack (que está cheia, porque armazena um máximo de cinco elementos). Observe que a função push retorna false quando ela não conseguir inserir um valor na pilha.¹

As linhas 27–28 invocam pop em um loop while para remover os cinco valores da pilha (note, na Figura 14.3, que os valores são realmente removidos na ordem último a entrar, primeiro a sair). Quando o driver tenta remover um sexto valor, a doubleStack está vazia, portanto o loop pop termina.

A linha 32 instancia a pilha do tipo inteiro intStack com a declaração

```
Stack< int > intStack;
```

(pronunciada como ‘intStack é uma Stack de int’). Como nenhum tamanho é especificado, o tamanho assume o padrão de 10 tal como especificado no construtor-padrão (Figura 14.2, linha 10). As linhas 37–41 fazem loop e invocam push para colocar valores sobre intStack até que ela esteja cheia, então as linhas 47–48 fazem loop e invocam pop para remover valores de intStack até que ela esteja vazia. Mais uma vez, note na saída que os valores são removidos na ordem último a entrar, primeiro a sair.

```

1  // Figura 14.3: fig14_03.cpp
2  // Programa de teste do template de classe Stack.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include "Stack.h" // definição de template de classe Stack
8
9  int main()
10 {
11     Stack< double > doubleStack( 5 ); // tamanho 5
12     double doubleValue = 1.1;
13
14     cout << "Pushing elements onto doubleStack\n";
15
16     // insere 5 doubles em doubleStack
17     while ( doubleStack.push( doubleValue ) )
18     {
19         cout << doubleValue << ' ';
20         doubleValue += 1.1;
21     } // fim do while
22
23     cout << "\nStack is full. Cannot push " << doubleValue
24         << "\n\nPopping elements from doubleStack\n";
25
26     // remove elementos de doubleStack
27     while ( doubleStack.pop( doubleValue ) )
28         cout << doubleValue << ' ';
29
30     cout << "\nStack is empty. Cannot pop\n";
31
32     Stack< int > intStack; // tamanho-padrão de 10
33     int intValue = 1;
34     cout << "\nPushing elements onto intStack\n";

```

Figura 14.3 Programa de teste do template de classe Stack.

(continua)

¹ A classe Stack (Figura 14.2) fornece a função isFull, que o programador pode utilizar para determinar se a pilha está cheia antes de tentar uma operação de inserção (push). Isso evitaria o erro potencial de inserir em uma pilha cheia. No Capítulo 16, “Tratamento de exceções”, se a operação não pudesse ser completada, a função push ‘lançaria uma exceção’. O programador pode escrever o código para ‘capturar’ essa exceção e, então, decidir como tratá-la apropriadamente para o aplicativo. A mesma técnica pode ser utilizada com a função pop em uma tentativa de remover um elemento de uma pilha vazia.

```

35
36     // insere 10 inteiros em intStack
37     while ( intStack.push( intValue ) )
38     {
39         cout << intValue << ' ';
40         intValue++;
41     } // fim do while
42
43     cout << "\nStack is full. Cannot push " << intValue
44         << "\n\nPopping elements from intStack\n";
45
46     // remove elementos de intStack
47     while ( intStack.pop( intValue ) )
48         cout << intValue << ' ';
49
50     cout << "\nStack is empty. Cannot pop" << endl;
51     return 0;
52 } // fim de main

```

Pushing elements onto doubleStack

1.1 2.2 3.3 4.4 5.5

Stack is full. Cannot push 6.6

Popping elements from doubleStack

5.5 4.4 3.3 2.2 1.1

Stack is empty. Cannot pop

Pushing elements onto intStack

1 2 3 4 5 6 7 8 9 10

Stack is full. Cannot push 11

Popping elements from intStack

10 9 8 7 6 5 4 3 2 1

Stack is empty. Cannot pop

Figura 14.3 Programa de teste do template de classe Stack.

(continuação)

*Criando templates de função para testar o template de classe **Stack< T >***

Note que o código na função `main` da Figura 14.3 é quase idêntico para as manipulações `doubleStack` nas linhas 11–30 e as manipulações `intStack` nas linhas 32–50. Isso apresenta outra oportunidade de utilizar um template de função. A Figura 14.4 define o template de função `testStack` (linhas 14–38) para realizar as mesmas tarefas que `main` na Figura 14.3 — inserir (push) uma série de valores sobre uma `Stack< T >` e remover (pop) os valores de uma `Stack< T >`. O template de função `testStack` utiliza o parâmetro de template `T`

```

1  // Figura 14.4: fig14_04.cpp
2  // Programa de teste do template de classe Stack. A função main utiliza um
3  // template de função para manipular objetos do tipo Stack< T >.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include <string>
9  using std::string;

```

Figura 14.4 Passando um objeto de template Stack para um template de função.

(continua)


```

10
11 #include "Stack.h" // definição de template de classe Stack
12
13 // template de função para manipular Stack< T >
14 template< typename T >
15 void testStack(
16     Stack< T > &theStack, // referência a Stack< T >
17     T value, // valor inicial a inserir
18     T increment, // incremento para valores subsequentes
19     const string stackName ) // nome do objeto Stack< T >
20 {
21     cout << "\nPushing elements onto " << stackName << '\n';
22
23     // insere elemento na Stack
24     while ( theStack.push( value ) )
25     {
26         cout << value << ' ';
27         value += increment;
28     } // fim do while
29
30     cout << "\nStack is full. Cannot push " << value
31         << "\n\nPopping elements from " << stackName << '\n';
32
33     // remove elementos da Stack
34     while ( theStack.pop( value ) )
35         cout << value << ' ';
36
37     cout << "\nStack is empty. Cannot pop" << endl;
38 } // fim do template de função testStack
39
40 int main()
41 {
42     Stack< double > doubleStack( 5 ); // tamanho 5
43     Stack< int > intStack; // tamanho-padrão de 10
44
45     testStack( doubleStack, 1.1, 1.1, "doubleStack" );
46     testStack( intStack, 1, 1, "intStack" );
47
48     return 0;
49 } // fim de main

```

Pushing elements onto doubleStack
1.1 2.2 3.3 4.4 5.5
Stack is full. Cannot push 6.6

Popping elements from doubleStack
5.5 4.4 3.3 2.2 1.1
Stack is empty. Cannot pop

Pushing elements onto intStack
1 2 3 4 5 6 7 8 9 10
Stack is full. Cannot push 11

Popping elements from intStack
10 9 8 7 6 5 4 3 2 1
Stack is empty. Cannot pop

Figura 14.4 Passando um objeto de template Stack para um template de função.

(continuação)

(especificado na linha 14) para representar o tipo de dados armazenado em `Stack< T >`. O template de função aceita quatro argumentos (linhas 16–19) — uma referência a um objeto de tipo `Stack< T >`, um valor de tipo `T` que será o primeiro valor inserido em `Stack< T >`, um valor de tipo `T` utilizado para incrementar os valores inseridos em `Stack< T >` e uma `string` que representa o nome do objeto `Stack< T >` para propósitos de saída. A função `main` (linhas 40–49) instancia um objeto do tipo `Stack< double >` chamado `doubleStack` (linha 42) e um objeto do tipo `Stack< int >` chamado `intStack` (linha 43) e os utiliza nas linhas 45 e 46. Todas as chamadas de função `testStack` resultam em uma especialização do template de função `testStack`. O compilador infere o tipo de `T` para `testStack` a partir do tipo utilizado para instanciar o primeiro argumento da função (isto é, o tipo utilizado para instanciar `doubleStack` ou `intStack`). A saída da Figura 14.4 coincide precisamente com saída da Figura 14.3.

14.5 Parâmetros sem tipo e tipos-padrão para templates de classes

O template de classe `Stack` da Seção 14.4 utilizou somente um parâmetro de tipo no cabeçalho de template (linha 6). Também é possível utilizar os **parâmetros de template não-tipo** ou **parâmetros não-tipo**, que podem ter argumentos-padrão e são tratados como `const`. Por exemplo, o cabeçalho de template poderia ser modificado para aceitar um parâmetro `int elements` como mostrado a seguir:

```
template< typename T, int elements > // elementos de parâmetro não-tipo
```

Então, uma declaração como

```
Stack< double, 100 > mostRecentSalesFigures;
```

poderia ser utilizada para instanciar (em tempo de compilação) uma especialização do template de classe `Stack` de 100 elementos de valores `double` chamados `mostRecentSalesFigures`; essa especialização de template de classe seria de tipo `Stack< double, 100 >`. O cabeçalho de classe então poderia conter um membro de dados `private` com uma declaração de array como

```
T stackHolder[ elements ]; // array para armazenar o conteúdo de Stack
```

Além disso, um parâmetro de tipo pode especificar um tipo-padrão. Por exemplo,

```
template< typename T = string > // por padrão, assume o tipo string
```

poderia especificar que uma `Stack` contém objetos `string` por padrão. Então, uma declaração como

```
Stack<> jobDescriptions;
```

poderia ser utilizada para instanciar uma especialização do template de classe `Stack` de strings chamadas `jobDescriptions`; essa especialização de template de classe seria do tipo `Stack< string >`. Os parâmetros de tipo-padrão devem ser os parâmetros mais à direita (finais) em uma lista de parâmetros de tipo de um template. Quando alguém instancia uma classe com dois ou mais tipos-padrão, se um tipo omitido não for o parâmetro de tipo mais à direita na lista de parâmetro de tipo, então os parâmetros de todos os tipos à direita desse tipo também devem ser omitidos.



Dica de desempenho 14.2

Quando apropriado, especifique o tamanho de uma classe contêiner (como uma classe de array ou de pilha) em tempo de compilação (possivelmente por um parâmetro de template não-tipo). Isso elimina o overhead de tempo de execução de utilizar `new` para criar o espaço dinamicamente.



Observação de engenharia de software 14.3

Especificar o tamanho de um contêiner em tempo de compilação evita o erro potencialmente fatal de tempo de execução se `new` não conseguir obter a memória necessária.

Nos exercícios, você será solicitado a utilizar um parâmetro não-tipo para criar um template para a nossa classe `Array` desenvolvida no Capítulo 11. Esse template permitirá que objetos `Array` sejam instanciados com um número especificado de elementos de um tipo especificado em tempo de compilação, em vez de criar espaço para os objetos `Array` em tempo de execução.

Em alguns casos, pode não ser possível utilizar um tipo particular com um template de classe. Por exemplo, o template `Stack` da Figura 14.2 requer que tipos definidos pelo usuário que serão armazenados em uma `Stack` devem fornecer um construtor-padrão e um operador de atribuição. Se um tipo definido pelo usuário particular não funcionar com nosso template `Stack` ou exigir processamento personalizado, você pode definir uma **especialização explícita** do template de classe para um tipo particular. Vamos supor que quiséssemos criar uma especialização explícita de `Stack` para os objetos `Employee`. Para fazer isso, forme uma nova classe com o nome `Stack< Employee >` como segue:

```
template<>
class Stack< Employee >
{
    // corpo da definição de classe
};
```

Observe que a especialização explícita `Stack< Employee >` é um substituto completo para o template de classe `Stack` que é específico ao tipo `Employee` — ela não utiliza nada do template original de classe e até pode ter membros diferentes.

14.6 Notas sobre templates e herança

Templates e herança se relacionam de diversas maneiras:

- Um template de classe pode ser derivado de uma especialização de template de classe.
- Um template de classe pode ser derivado de uma classe não-template.
- Uma especialização de template de classe pode ser derivada de uma especialização de template de classe.
- Uma classe não-template pode ser derivada de uma especialização de template de classe.

14.7 Notas sobre templates e friends

Vimos que as funções e classes inteiras podem ser declaradas como `friends` de classes não-template. Com templates de classe, a amizade pode ser estabelecida entre um template de classe e uma função global, uma função-membro de outra classe (possivelmente uma especialização de template de classe) ou mesmo uma classe inteira (possivelmente uma especialização de template de classe).

Por toda esta seção, supomos que definimos um template de classe para uma classe chamada `X` com um único parâmetro de tipo `T`, como em:

```
template< typename T > class X
```

Sob essa suposição, é possível tornar uma função `f1` friend de cada especialização de template de classe instanciada a partir do template de classe para a classe `X`. Para fazer isso, utilize uma declaração de amizade na forma

```
friend void f1();
```

Por exemplo, a função `f1` é friend de `X< double >`, `X< string >` e `X< Employee >` etc.

Também é possível tornar uma função `f2` friend apenas de uma especialização de template de classe com o mesmo argumento de tipo. Para fazer isso, utilize uma declaração de amizade na forma

```
friend void f2( X< T > & );
```

Por exemplo, se `T` for um `float`, a função `f2( X< float > & )` é friend da especialização de template de classe `X< float >`, mas não é friend da especificação de template de classe `X< string >`.

Você pode declarar que uma função-membro de outra classe seja uma amiga (friend) de qualquer especialização de template de classe gerada a partir do template de classe. Para fazer isso, a declaração friend deve qualificar o nome da função-membro de outra classe utilizando o nome de classe e o operador de resolução de escopo binário, como em:

```
friend void A::f3();
```

A declaração torna a função-membro `f3` da classe `A` friend de cada especialização de template de classe instanciada a partir do template de classe precedente. Por exemplo, a função `f3` da classe `A` é friend de `X< double >`, `X< string >` e `X< Employee >` etc.

Como com uma função global, a função-membro da outra classe só pode ser friend de uma especialização de template de classe com o mesmo argumento de tipo. Uma declaração de amizade na forma

```
friend void C< T >::f4( X< T > & );
```

para um tipo particular `T` como `float` torna função-membro

```
C< float >::f4( X< float > & )
```

uma função friend da especialização de template de classe `X< float >` *somente*.

Em alguns casos, é desejável tornar o conjunto de uma classe inteira de funções-membro friend de um template de classe. Nesse caso, uma declaração friend na forma

```
friend class Y;
```

torna cada função-membro da classe `Y` friend de cada especialização de template de classe produzida a partir do template de classe `X`.

Por fim, é possível tornar todas as funções-membro de uma especialização de template de classes friends de outra especialização de template de classe com o mesmo argumento de tipo. Por exemplo, uma declaração friend na forma:

```
friend class Z< T >;
```

indica que, quando uma especialização de template de classe é instanciada com um tipo particular para `T` (como `float`), todos os membros de `class Z< float >` tornam-se amigos da especialização de template de classe `X< float >`. Utilizamos esse relacionamento particular em vários exemplos do Capítulo 21, “Estruturas de dados”.

14.8 Notas sobre templates e membros static

E quanto aos membros de dados `static`? Lembre-se de que, com uma classe não-template, uma cópia de cada membro de dados `static` é compartilhada entre todos os objetos da classe, e o membro de dados `static` deve ser inicializado no escopo de arquivo.

Toda especialização de template de classe instanciada de um template de classe tem sua própria cópia de cada membro de dados `static` do template de classe; todos os objetos dessa especialização compartilham esse único membro de dados `static`. Além disso,

como ocorre com membros de dados `static` de classes não-template, os membros de dados `static` de especializações de template de classe devem ser definidos e, se necessário, inicializados no escopo de arquivo. Toda especialização de template de classe obtém sua própria cópia das funções-membro `static` do template de classe.

14.9 Síntese

Este capítulo introduziu um dos recursos mais poderosos do C++ — os templates. Você aprendeu a utilizar templates de função para permitir ao compilador produzir um conjunto de especializações de template de função que representam um grupo de funções sobrecarregadas relacionadas. Também discutimos como sobrecarregar um template de função para criar uma versão especializada de uma função que trata o processamento de um tipo de dados particular de maneira diferente das outras especializações de template de função. Em seguida, você aprendeu sobre templates de classe e especializações de template de classe. Você viu exemplos de como utilizar um template de classe para criar um grupo de tipos relacionados que realizam processamento idêntico em tipos de dados diferentes. Por fim, você aprendeu alguns dos relacionamentos entre templates, friends, herança e membros `static`.

No próximo capítulo, discutimos muitas das capacidades de E/S do C++ e demonstramos diversos manipuladores de fluxo que realizam várias tarefas de formatação.

Resumo

- Os templates permitem especificar uma série de funções relacionadas (sobrecarregadas) — chamadas especializações de template de função — ou uma série de classes relacionadas — chamadas especializações de template de classe.
- Para utilizar especializações de template de função, o programador escreve uma única definição de template de função. Com base nos tipos de argumento fornecidos em chamadas para essa função, o C++ gera especializações separadas para tratar cada tipo de chamada apropriadamente. Essas são compiladas junto com o restante do código-fonte de um programa.
- Todas as definições de template de função iniciam com a palavra-chave `template` seguida por parâmetros de template para o template de função entre colchetes angulares (`< >`); cada parâmetro de template que representa um tipo deve ser precedido pela palavra-chave `class` ou `typename`. As palavras-chave `typename` e `class` utilizadas para especificar parâmetros de template de função significam ‘qualquer tipo predefinido ou tipo definido pelo usuário’.
- Os parâmetros de template de definição de template são utilizados para especificar os tipos de argumentos para a função, o tipo de retorno da função e declarar as variáveis na função.
- O nome de um parâmetro de template pode ser declarado uma única vez na lista de parâmetros de tipo de um cabeçalho de template. Os nomes formais de parâmetro de tipo entre templates de função não precisam ser únicos.
- Um template de função pode ser sobrecarregado de várias maneiras. Podemos fornecer outros templates de função que especificam o mesmo nome de função mas parâmetros de função diferentes. Um template de função também pode ser sobrecarregado fornecendo a outras funções não-template o mesmo nome de função, mas parâmetros de função diferentes.
- Os templates de classe fornecem um meio de descrever uma classe genericamente e instanciar classes que são versões específicas de tipo dessa classe genérica.
- Os templates de classe são chamados tipos parametrizados; eles requerem parâmetros de tipos para especificar como personalizar um template de classe genérica a fim de formar uma especialização específica de template de classe.
- O programador que quer utilizar especializações de template de classe escreve um template de classe. Quando o programador precisa de uma nova classe de tipo específico, o programador utiliza uma notação concisa e o compilador escreve o código-fonte para a especialização de template de classe.
- Uma definição de template de classe é parecida com uma definição de classe convencional, exceto pelo fato de que é precedida por `template < typename T >` (ou `template < class T >`) para indicar que essa é uma definição de template de classe com parâmetro de tipo `T` que atua como um marcador de lugar para o tipo da classe a ser criada. O tipo `T` é mencionado por todo o cabeçalho de classe e definições de função-membro como um nome de tipo genérico.
- Todas as definições de membro-função fora de um template de classe iniciam com `template < typename T >` (ou `template < class T >`). Então, cada definição de função é parecida com uma definição de função convencional, exceto pelo fato de que os dados genéricos na classe são sempre listados genericamente como parâmetro de tipo `T`. O operador binário de resolução de escopo é utilizado com o nome de template de classe para associar cada definição de função-membro ao escopo do template de classe.
- É possível utilizar os parâmetros não-tipo no cabeçalho de uma classe ou template de função.
- Uma especialização explícita de um template de classe pode ser fornecida para sobrescrever um template de classe para um tipo específico.
- Um template de classe pode ser derivado de uma especialização de template de classe. Um template de classe pode ser derivado de uma classe não-template. Uma especialização de template de classe pode ser derivada de uma especialização de template de classe. Uma classe não-template pode ser derivada de uma especialização de template de classe.

- Funções e classes inteiras podem ser declaradas como *friends* de classes não-template. Com os templates de classe, os tipos óbvios de arranjos de amizade podem ser declarados. A amizade pode ser estabelecida entre um template de classe e uma função global, uma função-membro de outra classe (possivelmente uma especialização de template de classe) ou mesmo uma classe inteira (possivelmente uma especialização de template de classe).
- Cada especialização de template de classe instanciada a partir de um template de classe tem sua própria cópia de cada membro de dados `static` do template de classe; todos os objetos dessa especialização compartilham esse membro de dados `static`. E como ocorre com membros de dados `static` de classes não-template, os membros de dados `static` de especializações de template de classe devem ser definidos e, se necessário, inicializados no escopo de arquivo.
- Cada especialização de template de classe obtém uma cópia das funções-membro `static` do template de classe.

Terminologia

colchetes angulares (`< e >`)

`class`, palavra-chave, em um parâmetro de tipo de template

definição de template de classe

definição de template de função

especialização de template de classe

especialização de template de classe

especialização de template de função

especialização explícita

friend de um template

função-membro de um template de classe especialização

função-membro `static` de um template de classe

função-membro `static` de uma especialização de template de classe

macro

membro de dados `static` de um template de classe

membro de dados `static` de uma especialização de template de classe

parâmetro de template

parâmetro de template de tipo

parâmetro de template não-tipo

parâmetro de tipo

parâmetro não-tipo

programação genérica

sobrecarregar um template de função

template de classe

template de função

template, palavra-chave

`template< class T >`

`template< typename T >`

tipo parametrizado

`typename`

`typename`, palavra-chave

Exercícios de revisão

- 14.1** Determine quais das seguintes sentenças são *verdadeiras* e quais são *falsas*. Se uma sentença for *falsa*, explique por quê.
- Os parâmetros de template de uma definição de template de função são utilizados para especificar os tipos dos argumentos para a função, especificar o tipo de retorno da função e declarar as variáveis dentro da função.
 - As palavras-chave `typename` e `class` tal como utilizadas com um parâmetro de tipo de template significam, especificamente, ‘qualquer tipo de classe definido pelo usuário’.
 - Um template de função pode ser sobrecarregado por outro template de função com o mesmo nome de função.
 - Os nomes de parâmetro de template entre definições de template devem ser únicos.
 - Cada definição de função-membro fora de um template de classe deve iniciar com um cabeçalho de template.
 - Uma função *friend* de um template de classe deve ser uma especialização de template de função.
 - Se várias especializações de template de classe são geradas de um único template de classe com um único membro de dados `static`, cada uma das especializações de template de classe compartilha uma única cópia do membro de dados `static` do template de classe.
- 14.2** Preencha as lacunas em cada uma das seguintes sentenças:
- Os templates permitem especificar, com um único segmento de código, uma série inteira de funções relacionadas chamadas _____ ou uma série inteira de classes relacionadas chamadas _____.
 - Todas as definições de template de função iniciam com a palavra-chave _____, seguida por uma lista de parâmetros de template para o template de função entre _____.
 - Todas as funções relacionadas geradas a partir de um template de função têm o mesmo nome, então o compilador utiliza a solução _____ para invocar a função adequada.
 - Os templates de classe também são chamados de tipos _____.
 - O operador _____ é utilizado com um nome de template de classe para associar cada definição de função-membro ao escopo do template de classe.
 - Como ocorre com os membros de dados `static` de classes não-template, os membros de dados `static` de especializações de template de classe também devem ser definidos e, se necessário, inicializados no escopo de _____.

Respostas dos exercícios de revisão

- 14.1** a) Verdadeira. b) Falsa. As palavras-chave `typename` e `class` nesse contexto também permitem um parâmetro de tipo de um tipo predefinido. c) Verdadeira. d) Falsa. Os nomes de parâmetro de template entre templates de função não precisam ser únicos. e) Verdadeira. f) Falsa. Poderia ser uma função não-template. g) Falsa. Cada especialização de template de classe terá sua própria cópia do membro de dados `static`.
- 14.2** a) especializações de template de função, especializações de template de classe. b) `template`, colchetes angulares (`< e >`). c) sobrecarga. d) parametrizados. e) solução de escopo binário. f) arquivo.

Exercícios

- 14.3** Escreva um template de função `selectionSort` com base no programa de classificação da Figura 8.15. Escreva um driver que insere, classifica e gera saída de um array `int` e de um array `float`.
- 14.4** Sobrecarregue o template de função `printArray` da Figura 14.1 de modo que ele aceite dois argumentos do tipo inteiro adicionais, a saber, `int lowSubscript` e `int highSubscript`. Uma chamada para essa função imprimirá somente a parte designada do array. Valide `lowSubscript` e `highSubscript`; se qualquer uma estiver fora do intervalo ou se `highSubscript` for menor que ou igual a `lowSubscript`, a função sobrecarregada `printArray` deve retornar 0; caso contrário, `printArray` deve retornar o número de elementos impressos. A seguir, modifique `main` para exercitar ambas as versões de `printArray` nos arrays `a`, `b` e `c` (linhas 23–25 da Figura 14.1). Não deixe de testar todas as capacidades de ambas as versões de `printArray`.
- 14.5** Sobrecarregue o template de função `printArray` da Figura 14.1 com uma versão não-template que imprime especificamente um array de strings de caracteres em um formato tabular organizado.
- 14.6** Escreva um template de função simples para a função de predicado `isEqualTo` que compara seus dois argumentos do mesmo tipo com o operador de igualdade (`==`) e retorna `true` se eles forem iguais e `false` se não o forem. Utilize esse template de função em um programa que chama `isEqualTo` somente com uma variedade de tipos predefinidos. Agora escreva uma versão separada do programa que chama `isEqualTo` com um tipo de classe definido pelo usuário, mas que não sobrecarrega o operador de igualdade. O que acontece ao tentar executar esse programa? Agora sobrecarregue o operador de igualdade (com a função operador) `operator==`. Agora o que acontece ao tentar executar esse programa?
- 14.7** Use um parâmetro não-tipo de template `int numberOfElements` e um parâmetro de tipo `elementType` para ajudar a criar um template para a classe `Array` (figuras 11.6–11.7) que desenvolvemos no Capítulo 11. Esse template permitirá que os objetos `Array` sejam instanciados com um número especificado de elementos de um tipo de elemento especificado em tempo de compilação.
- 14.8** Escreva um programa com o template de classe `Array`. O template pode instanciar um `Array` de qualquer tipo de elemento. Sobrescreva o template com uma definição específica para um `Array` de elementos `float` (`class Array< float >`). O driver deve demonstrar a instanciação de um `Array` de `int` pelo template e deve mostrar que uma tentativa de instanciar um `Array` de `float` utiliza a definição fornecida em `class Array< float >`.
- 14.9** Diferencie os termos ‘template de função’ e ‘especialização de template de função’.
- 14.10** Qual deles se assemelha a um estêncil — o template de classe ou a especialização de template de classe? Explique sua resposta.
- 14.11** Qual é o relacionamento entre templates de função e sobrecarga?
- 14.12** Por que você escolheria utilizar um template de função em vez de uma macro?
- 14.13** Que problema de desempenho pode resultar do uso de templates de função e templates de classe?
- 14.14** O compilador realiza um processo de correspondência para determinar qual especialização de template de função chamar quando uma função é invocada. Sob quais circunstâncias uma tentativa de fazer uma correspondência resulta em um erro de compilação?
- 14.15** Por que é apropriado referir-se a um template de classe como um tipo parametrizado?
- 14.16** Explique por que um programa C++ utilizaria a instrução
- ```
Array< Employee > workerList(100);
```
- 14.17** Reveja sua resposta ao Exercício 14.16. Por que um programa C++ utilizaria a instrução
- ```
Array< Employee > workerList;
```
- 14.18** Explique o uso da seguinte notação em um programa C++:
- ```
template< typename T > Array< T >::Array(int s)
```
- 14.19** Por que você utilizaria um parâmetro não-tipo com um template de classe para um contêiner como um array ou pilha?
- 14.20** Descreva como fornecer uma especialização explícita de um template de classe.

**14.21** Descreva o relacionamento entre templates de classe e herança.

**14.22** Suponha que um template de classe tenha o cabeçalho

```
template< typename T > class Ct1
```

Descreva os relacionamentos de amizade estabelecidos colocando cada uma das seguintes declarações `friend` dentro desse template de classe. Os identificadores que iniciam com ‘f’ são funções, os que iniciam com ‘C’ são classes, os que iniciam com ‘Ct’ são templates de classe, e o T é um parâmetro de tipo de template (isto é, o T pode representar qualquer tipo fundamental ou tipo de classe).

- a) `friend void f1();`
- b) `friend void f2( Ct1< T > & );`
- c) `friend void C2::f3();`
- d) `friend void Ct3< T >::f4( Ct1< T > & );`
- e) `friend class C4;`
- f) `friend class Ct5< T >;`

**14.23** Suponha que o template de classe `Employee` tenha um membro de dados `static count`. Suponha que três especializações de template de classe são instanciadas a partir do template de classe. Quantas cópias do membro de dados `static` existirão? Como o uso de cada uma será restrito (se for de algum modo)?





*A consciência... não aparece dividida em partes... Um 'rio' ou um 'fluxo' são as metáforas pelas quais ela é mais naturalmente descrita.*  
William James

*Toda a notícia que seja digna de imprimir.*  
Adolph S. Ochs

*Não remova o marco divisório dos campos.*  
Amenchope

## Entrada/saída de fluxo

### OBJETIVOS

Neste capítulo, você aprenderá:

- Como utilizar entrada/saída de fluxo orientado a objetos do C++.
- Como formatar entrada e saída.
- A hierarquia de classes de E/S de fluxo.
- Como utilizar manipuladores de fluxo.
- Como controlar alinhamento e preenchimento.
- Como determinar o sucesso ou a falha de operações de entrada/saída.
- Como associar fluxos de saída aos fluxos de entrada.

- 15.1** Introdução
- 15.2** Fluxos
  - 15.2.1** Fluxos clássicos *versus* fluxos-padrão
  - 15.2.2** Arquivos de cabeçalho da biblioteca `iostream`
  - 15.2.3** Classes de entrada/saída de fluxo e objetos
- 15.3** Saída de fluxo
  - 15.3.1** Saída de variáveis `char *`
  - 15.3.2** Saída de caractere utilizando a função-membro `put`
- 15.4** Entrada de fluxo
  - 15.4.1** Funções-membro `get` e `getline`
  - 15.4.2** Funções-membro `peek`, `putback` e `ignore` de `istream`
  - 15.4.3** E/S fortemente tipada (*type-safe*)
- 15.5** E/S não formatada utilizando `read`, `write` e `gcount`
- 15.6** Introdução aos manipuladores de fluxos
  - 15.6.1** Base de fluxo integral: `dec`, `oct`, `hex` e `setbase`
  - 15.6.2** Precisão de ponto flutuante (`precision`, `setprecision`)
  - 15.6.3** Largura de campo (`width`, `setw`)
  - 15.6.4** Manipuladores de fluxo de saída definidos pelo usuário
- 15.7** Estados de formato de fluxo e manipuladores de fluxo
  - 15.7.1** Zeros finais e pontos de fração decimal (`showpoint`)
  - 15.7.2** Alinhamento (`left`, `right` e `internal`)
  - 15.7.3** Preenchimento (`fill`, `setfill`)
  - 15.7.4** Base de fluxo integral (`dec`, `oct`, `hex`, `showbase`)
  - 15.7.5** Números de ponto flutuante; notação científica e notação fixa (`scientific`, `fixed`)
  - 15.7.6** Controle de letras maiúsculas/minúsculas (`uppercase`)
  - 15.7.7** Especificando o formato booleano (`boolalpha`)
  - 15.7.8** Configurando e redefinindo o estado de formato via função-membro `flags`
- 15.8** Estados de erro de fluxo
- 15.9** Associando um fluxo de saída a um fluxo de entrada
- 15.10** Síntese

Resumo | Terminologia | Exercícios de revisão | Respostas dos exercícios de revisão | Exercícios

## 15.1 Introdução

As bibliotecas C++ padrão fornecem um extenso conjunto de capacidades de entrada/saída. Este capítulo discute uma série de capacidades suficientes para realizar a maioria das operações de E/S comuns e fornece uma visão geral das demais capacidades. Discutimos alguns desses recursos anteriormente no texto; agora fornecemos um tratamento mais completo. Muitos dos recursos de E/S que discutiremos são orientados a objetos. Esse estilo de E/S utiliza outros recursos do C++, como referências, sobrecarga de funções e sobrecarga de operadores.

O C++ utiliza **E/S fortemente tipada** (*type-safe I/O*). Toda operação de E/S é executada de maneira sensível ao tipo de dados. Se uma função-membro de E/S foi definida para tratar um tipo de dados particular, então essa função-membro é chamada para tratar esse tipo de dados. Se não houver nenhuma correspondência entre o tipo de dados reais e uma função para tratar esse tipo de dados, o compilador gera um erro. Portanto, dados inadequados não podem ‘mover-se dissimuladamente’ pelo sistema (como pode ocorrer em C, permitindo alguns erros estranhos e sutis).

Os usuários podem especificar como realizar E/S para objetos de tipos definidos pelo usuário sobrecarregando o operador de inserção de fluxo (`<<`) e o operador de extração de fluxo (`>>`). Essa **extensibilidade** é um dos mais valiosos recursos do C++.



### Observação de engenharia de software 15.1

Utilize a E/S no estilo C++ exclusivamente em programas C++, mesmo que a E/S no estilo C esteja disponível a programadores em C++.



### Dica de prevenção de erro 15.1

A E/S do C++ é fortemente tipada (type safe).



### Observação de engenharia de software 15.2

O C++ permite um tratamento comum de E/S para os tipos predefinidos e os tipos definidos pelo usuário. Esse aspecto comum facilita o desenvolvimento e a reutilização de software.

## 15.2 Fluxos

A E/S no C++ ocorre em **fluxos**, que são seqüências de bytes. Nas operações de entrada, os bytes fluem de um dispositivo (por exemplo, teclado, unidade de disco, conexão de rede) para a memória principal. Nas operações de saída, os bytes fluem da memória principal para um dispositivo (por exemplo, monitor, impressora, unidade de disco, conexão de rede etc.).

Um aplicativo associa significado com bytes. Os bytes poderiam representar caracteres, dados brutos, imagens gráficas, voz e vídeo digital ou quaisquer outras informações que um aplicativo possa exigir.

Os mecanismos de E/S de sistema devem transferir bytes de dispositivos para a memória (e vice-versa) de maneira consistente e confiável. Essas transferências freqüentemente envolvem algum movimento mecânico, como a rotação de um disco ou fita, ou pressionamentos de teclas em um teclado. O tempo que essas transferências levam é, em geral, muito maior que o tempo que o processador requer para manipular dados internamente. Portanto, as operações de E/S exigem planejamento e ajuste cuidadoso para assegurar um ótimo desempenho.

O C++ fornece capacidades de E/S tanto de ‘baixo nível’ como de ‘alto nível’. As capacidades de E/S de baixo nível (isto é, **E/S não formatada**) especificam que um número de bytes deve ser transferido do dispositivo para a memória ou da memória para o dispositivo. Nessas transferências, o byte individual é o item de interesse. Essas capacidades de baixo nível fornecem transferências de alta velocidade e alto volume, mas não são particularmente convenientes aos programadores.

Os programadores geralmente preferem uma visão de nível mais alto de E/S (isto é, **E/S formatada**), em que os bytes são agrupados em unidades significativas, como inteiros, números de ponto flutuante, caracteres, strings e tipos definidos pelo usuário. Essas capacidades orientadas a tipos são satisfatórias para a maior parte da E/S não relacionada com o processamento de arquivo de alto volume.



### Dica de desempenho 15.1

Utilize a E/S não formatada para obter o melhor desempenho no processamento de arquivo de alto volume.



### Dica de portabilidade 15.1

Utilizar a E/S não formatada pode levar a problemas de portabilidade, porque os dados não formatados não são portáveis em todas as plataformas.

### 15.2.1 Fluxos clássicos versus fluxos-padrão

No passado, as **bibliotecas de fluxo clássicas** do C++ permitiam a entrada e a saída de chars. Como ocupa apenas um byte, um char pode representar somente um conjunto limitado de caracteres (como os do conjunto de caracteres ASCII). Entretanto, muitos idiomas utilizam alfabetos que contêm mais caracteres do que aqueles que um byte char pode representar. O conjunto de caracteres ASCII não tem esses caracteres; o **conjunto de caracteres Unicode** tem. O Unicode é um extenso conjunto de caracteres internacional que representa a maioria dos idiomas comercialmente viáveis do mundo, símbolos matemáticos e muito mais. Para informações adicionais sobre o Unicode, visite [www.unicode.org](http://www.unicode.org).

O C++ inclui as **bibliotecas de fluxo padrão**, que permitem aos desenvolvedores construir sistemas capazes de realizar operações de E/S com caracteres Unicode. Para esse propósito, o C++ inclui um tipo adicional de caracteres chamado **wchar\_t**, que pode armazenar caracteres Unicode. O C++ padrão também reprojeteu as classes de fluxo clássico do C++, que processavam apenas chars, como templates de classe com especializações separadas para processar caracteres dos tipos char e wchar\_t, respectivamente. Utilizamos o tipo char de templates de classe com especializações separadas por todo este livro.

### 15.2.2 Arquivos de cabeçalho da biblioteca iostream

A biblioteca **iostream** do C++ fornece centenas de capacidades de E/S. Vários arquivos de cabeçalho contêm partes da interface da biblioteca.

A maioria dos programas C++ inclui o arquivo de cabeçalho `<iostream>`, que declara serviços básicos requeridos por todas as operações de E/S de fluxo. O arquivo de cabeçalho `<iostream>` define os objetos `cin`, `cout`, `cerr` e `clog`, que correspondem ao fluxo de entrada-padrão, ao fluxo de saída-padrão, ao fluxo de erro-padrão não armazenado em buffer e ao fluxo de erro armazenado em buffer, respectivamente. (Os objetos `cerr` e `clog` são discutidos na Seção 15.2.3.) Os serviços de E/S não formatada e formatada são fornecidos. O cabeçalho `<iomanip>` declara serviços úteis para realizar E/S formatada com os chamados **manipuladores de fluxo parametrizados**, como `setw` e `setprecision` e o cabeçalho `<fstream>` declara serviços para processamento de arquivo controlado pelo usuário. Utilizamos esse cabeçalho nos programas de processamento de arquivo do Capítulo 17.

As implementações do C++ geralmente contêm outras bibliotecas relacionadas a E/S que fornecem capacidades específicas do sistema, como o controle de dispositivos de uso especial para E/S de áudio e vídeo.

### 15.2.3 Classes de entrada/saída de fluxo e objetos

A biblioteca `iostream` fornece muitos templates para tratar operações comuns de E/S. Por exemplo, o template de classe `basic_istream` suporta operações de entrada de fluxo, o template da classe `basic_ostream` suporta operações de saída de fluxo e o template da classe `basic_iostream` suporta tanto operações de entrada de fluxo como de saída. Todo template tem uma especialização de template predefinida que permite a E/S de char. Além disso, a biblioteca `iostream` fornece um conjunto de typedefs que fornece aliases dessas especializações de template. O especificador **typedef** declara sinônimos (aliases) dos tipos de dados anteriormente definidos. Às vezes os programadores usam o typedef para criar nomes de tipo mais curtos ou mais legíveis. Por exemplo, a instrução

```
typedef Card *CardPtr;
```

define um nome de tipo adicional, `CardPtr`, como um sinônimo do tipo `Card *`. Note que criar um nome com typedef não cria um tipo de dados; typedef cria apenas um nome de tipo que pode ser utilizado no programa. A Seção 22.5 typedef discute typedef em detalhes. O typedef `istream` representa uma especialização de `basic_istream` que permite a entrada de char. Semelhantemente, o typedef `ostream` representa uma especialização de `basic_ostream` que permite a saída de char. O typedef `iostream` também representa uma especialização de `basic_iostream` que permite tanto a entrada como a saída de char. Usamos esses typedefs por todo este capítulo.

#### Hierarquia de templates de fluxo de E/S e sobrecarga de operadores

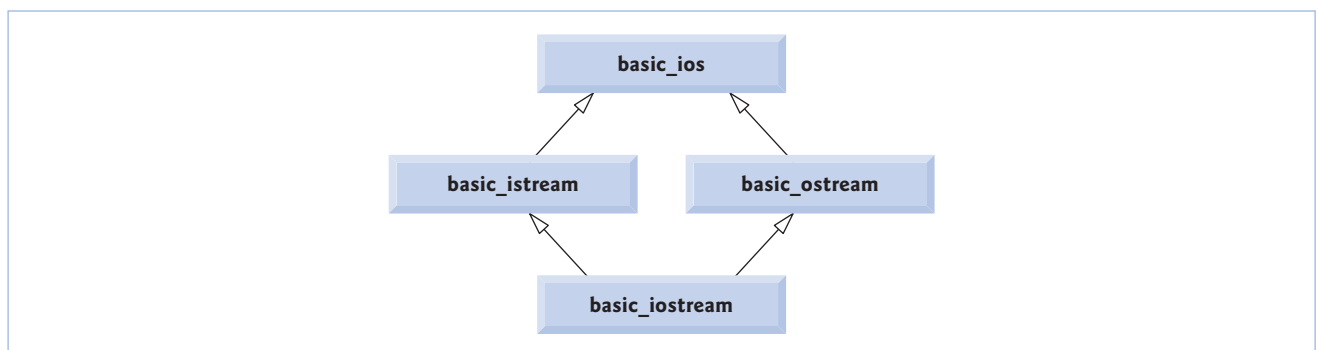
Os templates `basic_istream` e `basic_ostream` derivam ambos por herança simples do template de base `basic_ios`.<sup>1</sup> O template `basic_iostream` deriva por múltipla herança<sup>2</sup> dos templates `basic_istream` e `basic_ostream`. O diagrama de classe UML da Figura 15.1 resume esses relacionamentos de herança.

A sobrecarga de operadores fornece uma notação conveniente para realizar entrada/saída. O operador de deslocamento para a esquerda (`<<`) é sobrecarregado para designar a saída de fluxo e é referido como operador de inserção de fluxo. O operador de deslocamento para a direita (`>>`) é sobrecarregado para designar a entrada de fluxo e é referido como operador de extração de fluxo. Esses operadores são utilizados com os objetos de fluxo-padrão `cin`, `cout`, `cerr` e `clog` e, comumente, com objetos de fluxo definidos pelo usuário.

#### Objetos de fluxo padrão `cin`, `cout`, `cerr` e `clog`

O objeto predefinido `cin` é uma instância `istream` e dizemos que ele está ‘conectado’ (ou anexado) ao dispositivo de entrada-padrão, que normalmente é o teclado. O operador de extração de fluxo (`>>`), tal como utilizado na instrução a seguir, faz com que o valor da variável do tipo inteiro `grade` (supondo que `grade` foi declarada como uma variável `int`) seja inserido de `cin` para a memória:

```
cin >> grade; // os dados “fluem” na direção das setas
```



**Figura 15.1** Parte da hierarquia de templates de E/S de fluxo.

<sup>1</sup> Tecnicamente, os templates não herdam de outros templates. Entretanto, neste capítulo, discutimos apenas os templates no contexto das especializações de template que permitem a E/S de char. Essas especializações são classes e, assim, podem herdar umas das outras.

<sup>2</sup> A herança múltipla é discutida no Capítulo 24, “Outros tópicos”.

Observe que o compilador determina o tipo de dados de grade e seleciona o operador sobrecarregado de extração de fluxo apropriado. Supondo que grade foi declarada adequadamente, o operador de extração de fluxo não requer informações de tipo adicionais (como é o caso, por exemplo, na E/S no estilo C). O operador >> é sobrecarregado para realizar a entrada de itens de dados de tipos predefinidos, strings e valores de ponteiro.

O objeto predefinido cout é uma instância ostream e diz-se que ele está ‘conectado’ ao dispositivo de saída-padrão, que, normalmente, é a tela do vídeo. O operador de inserção de fluxo (<<), tal como utilizado na instrução a seguir, produz a saída do valor da variável grade da memória para o dispositivo de saída-padrão:

```
cout << grade; // os dados “fluem” na direção das setas
```

Observe que o compilador também determina o tipo de dados de grade (supondo que grade foi adequadamente declarada) e seleciona o operador de inserção de fluxo apropriado, portanto o operador de inserção de fluxo não requer informações de tipo adicionais. O operador << é sobrecarregado para gerar saída de itens de dados de tipos predefinidos, strings e valores de ponteiro.

O objeto predefinido cerr é uma instância ostream e diz-se que ele está ‘conectado’ ao dispositivo de erro-padrão. As saídas para o objeto cerr são **não armazenadas em buffer**, o que implica que cada inserção de fluxo para cerr faz com que sua saída apareça imediatamente — isso é apropriado para notificar prontamente o usuário dos erros.

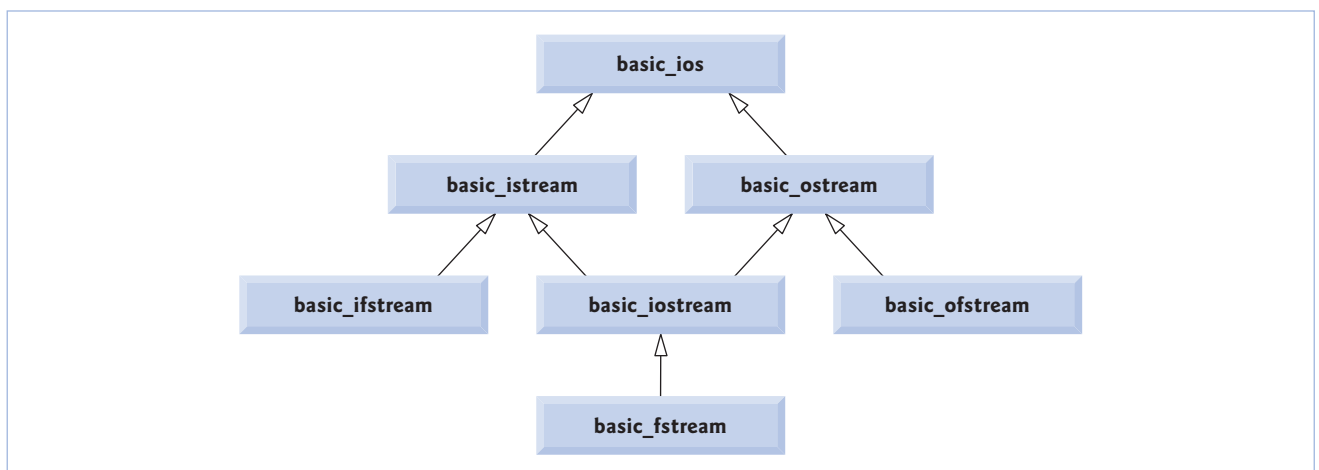
O objeto predefinido clog é uma instância da classe ostream e diz-se que ele está ‘conectado’ ao dispositivo de erro-padrão. As saídas para clog são **armazenadas em buffer**. Isso significa que cada inserção em clog poderia fazer com que sua saída ficasse presa a um buffer até que este fosse preenchido ou esvaziado. O armazenamento em buffer é uma técnica de aprimoramento de desempenho de E/S discutida em cursos de sistemas operacionais.

### Templates de processamento de arquivo

O processamento de arquivo C++ utiliza templates da classe **basic\_ifstream** (para entrada de arquivo), **basic\_ofstream** (para saída de arquivo) e **basic\_fstream** (para entrada e saída de arquivo). Cada template de classe tem uma especialização de template predefinida que permite a E/S de char. O C++ fornece um conjunto de typedefs que fornece aliases para essas especializações de template. Por exemplo, o typedef **ifstream** representa uma especialização de **basic\_ifstream** que permite a entrada de char a partir de um arquivo. De modo semelhante, typedef **ofstream** representa uma especialização de **basic\_ofstream** que permite a saída de char para um arquivo. Além disso, typedef **fstream** representa uma especialização de **basic\_fstream** que permite a entrada e a saída de char de e para um arquivo. O template **basic\_ifstream** herda de **basic\_istream**, **basic\_ofstream** herda de **basic\_ostream**, e **basic\_fstream** herda de **basic\_iostream**. O diagrama de classe UML da Figura 15.2 resume os vários relacionamentos de herança das classes relacionadas com E/S. A hierarquia de classes de E/S de fluxo fornece a maioria das capacidades de que os programadores precisam. Consulte a referência de biblioteca de classes do sistema C++ para obter informações adicionais sobre o processamento de arquivo.

## 15.3 Saída de fluxo

As capacidades de saída formatadas e não formatadas são fornecidas por ostream. As capacidades de saída incluem saída de tipos de dados-padrão com o operador de inserção de fluxo (<<); saída de caracteres via função-membro put; saída não formatada via função-membro write (Seção 15.5); saída de inteiros nos formatos decimal, octal e hexadecimal (Seção 15.6.1); saída de valores de ponto flutuante com várias precisões (Seção 15.6.2), com pontos forçados de fração decimal (Seção 15.7.1), em notação científica e fixa (Seção 15.7.5); saída de dados alinhados em campos de larguras designadas (Seção 15.7.2); saída de dados em campos preenchidos com caracteres especificados (Seção 15.7.3); e saída de letras maiúsculas em notação científica e hexadecimal (Seção 15.7.6).



**Figura 15.2** Parte da hierarquia de template de E/S de fluxo mostrando os principais templates de processamento de arquivo.

### 15.3.1 Saída de variáveis char \*

O C++ determina os tipos de dados automaticamente, uma melhora em relação ao C. Infelizmente, esse recurso às vezes ‘atrapalha’. Por exemplo, suponha que quiséssemos imprimir o valor de um `char *` para uma string de caracteres (isto é, o endereço de memória do primeiro caractere dessa string). Entretanto, o operador `<<` foi sobrecarregado para imprimir dados de tipo `char *` como uma string terminada por caractere nulo. A solução é fazer coerção do `char *` em um `void *` (de fato, isso deve ser feito com qualquer variável de ponteiro que o programador deseja enviar para a saída como um endereço). A Figura 15.3 demonstra a impressão de uma variável `char *` nos formatos de string e de endereço. Observe que o endereço é impresso como um número hexadecimal (base 16). [Nota: O leitor que quiser aprender mais sobre números hexadecimais deve ler o Apêndice D, “Sistemas de numeração”.] Falaremos mais sobre o controle das bases de números nas seções 15.6.1, 15.7.4, 15.7.5 e 15.7.7. [Nota: O endereço de memória mostrado na saída do programa da Figura 15.3 pode diferir entre compiladores.]

### 15.3.2 Saída de caractere utilizando a função-membro put

Podemos utilizar a função-membro `put` para gerar a saída de caracteres. Por exemplo, a instrução

```
cout.put('A');
```

exibe um único caractere A. As chamadas para `put` podem ser em cascata, como na instrução

```
cout.put('A').put('\n');
```

que gera a saída da letra A seguida por um caractere de nova linha. Como ocorre com `<<`, a instrução precedente executa dessa maneira, porque o operador de ponto (.) avalia da esquerda para a direita e a função-membro `put` retorna uma referência ao objeto `ostream` (`cout`) que recebeu a chamada `put`. A função `put` também pode ser chamada com uma expressão numérica que representa um valor ASCII, como na seguinte instrução

```
cout.put(65);
```

que também gera saída de A.

## 15.4 Entrada de fluxo

Agora consideremos a entrada de fluxo. As capacidades de entrada formatada e não formatada são fornecidas por `istream`. O operador de extração de fluxo (isto é, o operador `>>` sobrecarregado) normalmente pula os **caracteres de espaço em branco** (como espaços, tabulações e nova linha) no fluxo de entrada; mais adiante, veremos como mudar esse comportamento. Depois de cada entrada, o operador de extração de fluxo retorna uma referência ao objeto fluxo que recebeu a mensagem de extração (por exemplo, `cin` na expressão `cin >> grade`). Se essa referência for utilizada como uma condição (por exemplo, em uma condição de continuação do loop da instrução `while`), a função operadora de coerção `void *` sobrecarregada do fluxo é invocada implicitamente para converter a referência em um valor de ponteiro não-nulo ou nulo com base no sucesso ou falha da última operação de entrada. Para indicar sucesso, um ponteiro não-

```

1 // Figura 15.3: Fig15_03.cpp
2 // Imprimindo o endereço armazenado em uma variável char *.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9 char *word = "again";
10
11 // exibe o valor de char * e, então, o valor de char *
12 // static_cast para void *
13 cout << "Value of word is: " << word << endl
14 << "Value of static_cast< void * >(word) is: "
15 << static_cast< void * >(word) << endl;
16 return 0;
17 } // fim de main
```

```
Value of word is: again
Value of static_cast< void * >(word) is: 00428300
```

**Figura 15.3** Imprimindo o endereço armazenado em uma variável `char *`.

nulo é convertido no valor `bool true` e, para indicar falha, o ponteiro nulo é convertido no valor `bool false`. Quando uma tentativa de ler além do fim de um fluxo é feita, o operador de coerção `void *` sobrecarregado do fluxo retorna o ponteiro nulo para indicar o fim do arquivo.

Todo objeto de fluxo contém um conjunto de **bits de estado** utilizado para controlar o estado do fluxo (isto é, formatar, configurar estados de erro etc.). Esses bits são utilizados pelo operador de coerção `void *` sobrecarregado do fluxo para determinar se ele retorna um ponteiro não-nulo ou o ponteiro nulo. A extração de fluxo faz com que o **failbit** do fluxo seja configurado se os dados do tipo errado forem inseridos e faz com que **badbit** do fluxo seja configurado se a operação falhar. As seções 15.7 e 15.8 discutem os bits de estado do fluxo em detalhes e, então, mostram como testar esses bits depois de uma operação de E/S.

### 15.4.1 Funções-membro `get` e `getline`

A função-membro `get` sem argumentos insere um caractere a partir do fluxo designado (incluindo caracteres de espaço em branco e outros caracteres não gráficos, como a sequência-chave que representa o fim do arquivo) e o retorna como o valor da chamada de função. Essa versão de `get` retorna EOF quando o fim do arquivo é encontrado no fluxo.

#### Utilizando as funções-membro `eof`, `get` e `put`

A Figura 15.4 demonstra o uso de funções-membro `eof` e `get` no fluxo de entrada `cin` e a função-membro `put` no fluxo de saída `cout`. O programa primeiro imprime o valor de `cin.eof()` — isto é, `false` (0 na saída) — para mostrar que o fim do arquivo não ocorreu em `cin`. O usuário insere uma linha de texto e pressiona *Enter* seguido do fim do arquivo (`<ctrl>-z` em sistemas Microsoft Windows, `<ctrl>-d` em sistemas UNIX e Macintosh). A linha 17 lê cada caractere, que a linha 18 envia para a saída com `cout` utilizando a função-membro `put`. Quando o fim do arquivo é encontrado, a instrução `while` termina, e a linha 22 exibe o valor de `cin.eof()`, que agora é

```

1 // Figura 15.4: Fig15_04.cpp
2 // Utilizando funções-membro get, put e eof.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6 using std::endl;
7
8 int main()
9 {
10 int character; // utiliza int, porque char não pode representar EOF
11
12 // solicita para o usuário inserir linha de texto
13 cout << "Before input, cin.eof() is " << cin.eof() << endl
14 << "Enter a sentence followed by end-of-file:" << endl;
15
16 // utiliza get para ler cada caractere; utiliza put para exibi-los
17 while ((character = cin.get()) != EOF)
18 cout.put(character);
19
20 // exibe caractere de fim do arquivo
21 cout << "\nEOF in this system is: " << character << endl;
22 cout << "After input of EOF, cin.eof() is " << cin.eof() << endl;
23 return 0;
24 } // fim de main

```

```

Before input, cin.eof() is 0
Enter a sentence followed by end-of-file:
Testing the get and put member functions
Testing the get and put member functions
^Z

```

```

EOF in this system is: -1
After input of EOF, cin.eof() is 1

```

**Figura 15.4** Funções-membro `get`, `put` e `eof`.



true (1 na saída), para mostrar que o fim do arquivo foi configurado em cin. Observe que esse programa utiliza a versão de função-membro istream get que não aceita argumentos e retorna o caractere sendo inserido (linha 17). A função eof retorna true somente depois que o programa tenta ler além do último caractere no fluxo.

A função-membro get com um argumento de referência de caractere insere o próximo caractere a partir do fluxo de entrada (mesmo se este for um caractere de espaço em branco) e o armazena no argumento de caractere. Essa versão de get retorna uma referência para o objeto istream para o qual a função-membro get está sendo invocada.

Uma terceira versão de get aceita três argumentos — um array de caracteres, um limite de tamanho e um delimitador (com o valor padrão '\n'). Essa versão lê caracteres do fluxo de entrada. Ela lê um número a menos que o número de caracteres máximo especificado e termina, ou termina logo que o delimitador é lido. Um caractere nulo é inserido para terminar a string de entrada no array de caracteres utilizado como um buffer pelo programa. O delimitador não é colocado no array de caracteres, mas permanece no fluxo de entrada (o delimitador será o próximo caractere lido). Portanto, o resultado de um segundo get consecutivo é uma linha vazia, a menos que o caractere delimitador seja removido do fluxo de entrada (possivelmente com cin.ignore()).

### Comparando cin e cin.get

A Figura 15.5 compara a entrada usando extração de fluxo com cin (que lê caracteres até um caractere de espaço em branco ser encontrado) e a entrada usando cin.get. Observe que a chamada para cin.get (linha 24) não especifica um delimitador, então o caractere '\n' padrão é utilizado.

### Utilizando a função-membro getline

A função-membro getline opera de maneira semelhante à terceira versão da função-membro get e insere um caractere nulo depois da linha no array de caracteres. A função getline remove o delimitador do fluxo (isto é, lê o caractere e o descarta), mas não o armazena no array de caracteres. O programa da Figura 15.6 demonstra o uso da função-membro getline para inserir uma linha de texto (linha 15).

```

1 // Figura 15.5: Fig15_05.cpp
2 // Contrastando a entrada de uma string via cin e cin.get.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6 using std::endl;
7
8 int main()
9 {
10 // cria dois arrays de char, cada um com 80 elementos
11 const int SIZE = 80;
12 char buffer1[SIZE];
13 char buffer2[SIZE];
14
15 // utiliza cin para inserir caracteres em buffer1
16 cout << "Enter a sentence:" << endl;
17 cin >> buffer1;
18
19 // exibe o conteúdo de buffer1
20 cout << "\nThe string read with cin was:" << endl
21 << buffer1 << endl << endl;
22
23 // usa cin.get para inserir caracteres em buffer2
24 cin.get(buffer2, SIZE);
25
26 // exibe o conteúdo buffer2
27 cout << "The string read with cin.get was:" << endl
28 << buffer2 << endl;
29 return 0;
30 } // fim de main

```

**Figura 15.5** A entrada de uma string utilizando cin com extração de fluxo contrastada com a entrada utilizando cin.get.

(continua)

```

Enter a sentence:
Contrasting string input with cin and cin.get

The string read with cin was:
Contrasting

The string read with cin.get was:
string input with cin and cin.get

```

**Figura 15.5** A entrada de uma string utilizando `cin` com extração de fluxo contrastada com a entrada utilizando `cin.get`. (continuação)

```

1 // Figura 15.6: Fig15_06.cpp
2 // Inserindo caracteres utilizando a função-membro cin getline.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6 using std::endl;
7
8 int main()
9 {
10 const int SIZE = 80;
11 char buffer[SIZE]; // cria array de 80 caracteres
12
13 // insere caracteres no buffer via função cin getline
14 cout << "Enter a sentence:" << endl;
15 cin.getline(buffer, SIZE);
16
17 // exibe o conteúdo do buffer
18 cout << "\nThe sentence entered is:" << endl << buffer << endl;
19 return 0;
20 } // fim de main

```

```

Enter a sentence:
Using the getline member function

The sentence entered is:
Using the getline member function

```

**Figura 15.6** Inserindo dados de caracteres com a função-membro `cin getline`.

### 15.4.2 Funções-membro `peek`, `putback` e `ignore` de `istream`

A função-membro `ignore` de `istream` lê e descarta um número designado de caracteres (o padrão é um caractere) ou termina ao encontrar um delimitador designado (o delimitador-padrão é EOF, que faz com que `ignore` pule para o fim do arquivo durante a leitura de um arquivo).

A função-membro `putback` coloca o caractere anterior obtido por um `get` a partir de um fluxo de entrada de volta nesse fluxo. Essa função é útil para aplicativos que varrem um fluxo de entrada procurando por um campo que inicia com um caractere específico. Quando esse caractere é inserido, o aplicativo retorna o caractere para o fluxo, então o caractere pode ser incluído nos dados de entrada.

A função-membro `peek` retorna o próximo caractere de um fluxo de entrada, mas não remove o caractere do fluxo.

### 15.4.3 E/S fortemente tipada (*type-safe*)

O C++ oferece E/S fortemente tipada (*type-safe*). Os operadores `<<` e `>>` são sobrecarregados para aceitar itens de dados de tipos específicos. Se dados inesperados são processados, vários bits de erro são configurados, o que o usuário pode testar para determinar se uma operação de E/S foi ou não bem-sucedida. Se o operador `<<` não foi sobrecarregado para um tipo definido pelo usuário e você tentar realizar entrada ou saída do conteúdo de um objeto desse tipo definido pelo usuário, o compilador informa um erro. Isso permite ao programa ‘permanecer no controle’. Discutimos esses estados de erro na Seção 15.8.

## 15.5 E/S não formatada utilizando read, write e gcount

A entrada/saída não formatada é realizada utilizando as funções-membro `read` e `write` de `istream` e `ostream`, respectivamente. A função-membro `read` lê algum número de bytes em um array de caracteres na memória; a função-membro `write` gera uma saída de bytes de um array de caracteres. Esses bytes não são formatados de nenhuma maneira. A entrada ou saída desses bytes é realizada como bytes brutos. Por exemplo, a chamada

```
char buffer[] = "HAPPY BIRTHDAY";
cout.write(buffer, 10);
```

gera saída dos primeiros 10 bytes de `buffer` (incluindo os caracteres nulos, se houver algum, que fariam com que a saída com `cout` e `<<` terminasse). A chamada

```
cout.write("ABCDEFGHJKLMNOPQRSTUVWXYZ", 10);
```

exibe os 10 primeiros caracteres do alfabeto.

A função-membro `read` insere um número designado de caracteres em um array de caracteres. Se um número de caracteres menor do que o designado for lido, `failbit` é configurado. A Seção 15.8 mostra como determinar se `failbit` foi ou não configurado. A função-membro `gcount` informa o número de caracteres lidos pela última operação de entrada.

A Figura 15.7 demonstra as funções-membro `istream` `read` e `gcount`, e a função-membro `ostream` `write`. O programa insere 20 caracteres (de uma sequência de entrada mais longa) no array de caracteres `buffer` com `read` (linha 15), determina o número de entrada de caracteres com `gcount` (linha 19) e gera saída dos caracteres em `buffer` com `write` (linha 19).

## 15.6 Introdução aos manipuladores de fluxos

O C++ fornece vários **manipuladores de fluxo** que realizam as tarefas de formatação. Os manipuladores de fluxo fornecem capacidades como configurar larguras de campo, configurar precisão, configurar e redefinir o estado de formatação, configurar o preenchimento de caracteres em campos, esvaziar fluxos, inserir uma nova linha no fluxo de saída (e esvaziar o fluxo), inserir um caractere nulo no fluxo de saída e pular espaço em branco no fluxo de entrada. Esses recursos são descritos nas seções a seguir.

```
1 // Figura 15.7: Fig15_07.cpp
2 // E/S não formatada utilizando read, gcount e write.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6 using std::endl;
7
8 int main()
9 {
10 const int SIZE = 80;
11 char buffer[SIZE]; // cria array de 80 caracteres
12
13 // utiliza função read para inserir caracteres no buffer
14 cout << "Enter a sentence:" << endl;
15 cin.read(buffer, 20);
16
17 // utiliza funções write e gcount para exibir caracteres de buffer
18 cout << endl << "The sentence entered was:" << endl;
19 cout.write(buffer, cin.gcount());
20 cout << endl;
21 return 0;
22 } // fim de main
```

Enter a sentence:

**Using the read, write, and gcount member functions**

The sentence entered was:

Using the read, writ

**Figura 15.7** E/S não formatada utilizando as funções-membro `read`, `gcount` e `write`.

### 15.6.1 Base de fluxo integral: dec, oct, hex e setbase

Os inteiros são interpretados normalmente como valores decimais (base 10). Para alterar a base em que os inteiros são interpretados em um fluxo, insira o manipulador **hex** para configurar a base como hexadecimal (base 16) ou insira o manipulador **oct** para configurá-la como octal (base 8). Insira o manipulador **dec** para redefinir a base de fluxo como decimal.

A base de um fluxo também pode ser alterada pelo manipulador de fluxo **setbase**, que aceita um argumento de inteiro de 10, 8 ou 16 para configurar a base como decimal, octal ou hexadecimal, respectivamente. Como **setbase** aceita um argumento, é chamado de manipulador de fluxo parametrizado. Utilizar **setbase** (ou qualquer outro manipulador parametrizado) requer a inclusão do arquivo de cabeçalho `<iomanip>`. O valor de base do fluxo permanece o mesmo até ser alterado explicitamente; as configurações **setbase** são ‘aderentes’. A Figura 15.8 demonstra os manipuladores de fluxo **hex**, **oct**, **dec** e **setbase**.

### 15.6.2 Precisão de ponto flutuante (precision, setprecision)

Podemos controlar a **precisão** de números de ponto flutuante (isto é, o número de dígitos à direita do ponto de fração decimal) utilizando o manipulador de fluxo **setprecision** ou a função-membro **precision** de `ios_base`. Uma chamada a qualquer um desses configura a precisão de todas as operações de saída subsequentes até a próxima chamada de configuração de precisão. Uma chamada à função-membro **precision** sem argumentos retorna a configuração de precisão atual (isso é o que você precisa utilizar para poder restaurar

```

1 // Figura 15.8: Fig15_08.cpp
2 // Utilizando manipuladores de fluxo hex, oct, dec e setbase.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6 using std::dec;
7 using std::endl;
8 using std::hex;
9 using std::oct;
10
11 #include <iomanip>
12 using std::setbase;
13
14 int main()
15 {
16 int number;
17
18 cout << "Enter a decimal number: ";
19 cin >> number; // insere número
20
21 // utiliza manipulador de fluxo hex para mostrar o número hexadecimal
22 cout << number << " in hexadecimal is: " << hex
23 << number << endl;
24
25 // utiliza manipulador de fluxo oct para mostrar número octal
26 cout << dec << number << " in octal is: "
27 << oct << number << endl;
28
29 // utiliza manipulador de fluxo setbase para mostrar número decimal
30 cout << setbase(10) << number << " in decimal is: "
31 << number << endl;
32 return 0;
33 } // fim de main

```

```

Enter a decimal number: 20
20 in hexadecimal is: 14
20 in octal is: 24
20 in decimal is: 20

```

**Figura 15.8** Manipuladores de fluxo **hex**, **oct**, **dec** e **setbase**.

a precisão original ao final depois que uma configuração ‘aderente’ não é mais necessária). O programa da Figura 15.9 utiliza tanto uma função-membro `precision` (linha 28) como o manipulador `setprecision` (linha 37) para imprimir uma tabela que mostra a raiz quadrada de 2, com precisão variando de 0–9.

### 15.6.3 Largura de campo (width, setw)

A função-membro `width` (da classe básica `ios_base`) configura a largura de campo (isto é, o número de posições de caractere que a saída de um valor deve ter ou o número máximo de caracteres que deve ser inserido) e retorna a largura anterior. Se os valores enviados para a saída forem mais estreitos que a largura de campo, **caracteres de preenchimento** são inseridos como **preenchimento**. Um valor maior que o da largura designada não será truncado — o número inteiro será impresso. A função `width` sem argumentos retorna a configuração atual.

```

1 // Figura 15.9: Fig15_09.cpp
2 // Controlando a precisão de valores de ponto flutuante.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::fixed;
7
8 #include <iomanip>
9 using std::setprecision;
10
11 #include <cmath>
12 using std::sqrt; // protótipo de sqrt
13
14 int main()
15 {
16 double root2 = sqrt(2.0); // calcula a raiz quadrada de 2
17 int places; // precisão, varia de 0-9
18
19 cout << "Square root of 2 with precisions 0-9." << endl
20 << "Precision set by ios_base member function "
21 << "precision:" << endl;
22
23 cout << fixed; // usa notação de ponto fixo
24
25 // exibe raiz quadrada utilizando a precisão da função ios_base
26 for (places = 0; places <= 9; places++)
27 {
28 cout.precision(places);
29 cout << root2 << endl;
30 } // fim do for
31
32 cout << "\nPrecision set by stream manipulator "
33 << "setprecision:" << endl;
34
35 // configura a precisão de cada dígito, então exibe a raiz quadrada
36 for (places = 0; places <= 9; places++)
37 cout << setprecision(places) << root2 << endl;
38
39 return 0;
40 } // fim de main

```

Figura 15.9 Precisão de valores de ponto flutuante.

(continua)

```

Square root of 2 with precisions 0-9.
Precision set by ios_base member function precision:
1
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562

Precision set by stream manipulator setprecision:
1
1.4
1.41
1.414
1.4142
1.41421
1.414214
1.4142136
1.41421356
1.414213562

```

**Figura 15.9** Precisão de valores de ponto flutuante.

(continuação)



### Erro comum de programação 15.1

*A configuração de largura se aplica somente à próxima inserção ou extração (isto é, a configuração de largura não é ‘aderente’); depois, a largura é configurada implicitamente como 0 (isto é, a entrada e saída serão realizadas com configurações-padrão). Supor que a configuração de largura se aplica a todas as saídas subsequentes é um erro de lógica.*



### Erro comum de programação 15.2

*Quando um campo não for suficientemente largo para tratar saídas, estas são impressas com a largura necessária, o que pode resultar em saídas confusas.*

A Figura 15.10 demonstra o uso da função-membro `width` tanto na entrada como na saída. Observe que, na entrada de um array `char`, será lido um máximo de um caractere a menos que a largura porque se guardou espaço para o caractere nulo a ser colocado na string de entrada. Lembre-se de que a extração de fluxo termina quando o espaço em branco não inicial é encontrado. O manipulador de fluxo `setw` também pode ser utilizado para configurar a largura de campo.

[Nota: Quando solicitado a fornecer a entrada na Figura 15.10, o usuário deve inserir uma linha de texto e pressionar *Enter* seguido pelo fim do arquivo (`<ctrl>-z` em sistemas Microsoft Windows, `<ctrl>-d` em sistemas UNIX e Macintosh).]

```

1 // Figura 15.10: Fig15_10.cpp
2 // Demonstrando a função-membro width.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6 using std::endl;
7
8 int main()

```

**Figura 15.10** Função-membro `width` da classe `ios_base`.

(continua)

```

9 {
10 int widthValue = 4;
11 char sentence[10];
12
13 cout << "Enter a sentence:" << endl;
14 cin.width(5); // insere somente 5 caracteres da frase
15
16 // configura largura de campo, então exibe caracteres com base nessa largura
17 while (cin >> sentence)
18 {
19 cout.width(widthValue++);
20 cout << sentence << endl;
21 cin.width(5); // insere mais 5 caracteres a partir da frase
22 } // fim do while
23
24 return 0;
25 } // fim de main

```

```

Enter a sentence:
This is a test of the width member function
This
 is
 a
 test
 of
 the
 width
 h
 memb
 er
 func
 tion

```

Figura 15.10 Função-membro width da classe ios\_base.

(continuação)

### 15.6.4 Manipuladores de fluxo de saída definidos pelo usuário

Os programadores podem criar seus próprios manipuladores.<sup>3</sup> A Figura 15.11 mostra a criação e o uso de novos manipuladores de fluxo não parametrizados `bell` (linhas 10–13), `carriageReturn` (linhas 16–19), `tab` (linhas 22–25) e `endLine` (linhas 29–32). Para manipuladores de fluxo de saída, o tipo de retorno e o parâmetro devem ser do tipo `ostream &`. Quando a linha 37 insere o manipulador no fluxo de saída `endLine`, a função `endLine` é chamada e a linha 31 envia a sequência de escape `\n` e o manipulador `flush` para o fluxo de saída-padrão `cout`. De modo semelhante, quando as linhas 37–46 inserem os manipuladores `tab`, `bell` e `carriageReturn` no fluxo de saída, suas funções correspondentes — `tab` (linha 22), `bell` (linha 10) e `carriageReturn` (linha 16) são chamadas, as quais, por sua vez, geram saída de várias sequências de escape.

## 15.7 Estados de formato de fluxo e manipuladores de fluxo

Vários manipuladores de fluxo podem ser utilizados para especificar os tipos de formatação a ser realizados durante as operações de E/S de fluxo. Os manipuladores de fluxo controlam as configurações de formato de saída. A Figura 15.12 lista cada manipulador de fluxo que controla o estado de formato de um dado fluxo. Todos esses manipuladores pertencem à classe `ios_base`. Mostramos exemplos da maioria desses manipuladores de fluxo nas próximas seções.

<sup>3</sup> Os programadores também podem criar seus próprios manipuladores de fluxo parametrizados — consulte a documentação do C++ do seu compilador para obter instruções sobre como fazer isso.



```

1 // Figura 15.11: Fig15_11.cpp
2 // Criando e testando manipuladores de fluxo não parametrizados
3 // definidos pelo usuário.
4 #include <iostream>
5 using std::cout;
6 using std::flush;
7 using std::ostream;
8
9 // manipulador bell (usando a sequência de escape \a)
10 ostream& bell(ostream& output)
11 {
12 return output << '\a'; // emite bip de sistema
13 } // fim do manipulador bell
14
15 // manipulador carriageReturn (usando a sequência de escape \r)
16 ostream& carriageReturn(ostream& output)
17 {
18 return output << '\r'; // emite retorno de carro
19 } // fim do manipulador carriageReturn
20
21 // manipulador tab (usando a sequência de escape \t)
22 ostream& tab(ostream& output)
23 {
24 return output << '\t'; // emite tab
25 } // fim do manipulador tab
26
27 // manipulador endLine (usando a sequência de escape \n e a
28 // função-membro flush)
29 ostream& endLine(ostream& output)
30 {
31 return output << '\n' << flush; // emite fim de linha do tipo endl
32 } // fim do manipulador endLine
33
34 int main()
35 {
36 // usa manipuladores tab e endLine
37 cout << "Testing the tab manipulator:" << endLine
38 << 'a' << tab << 'b' << tab << 'c' << endLine;
39
40 cout << "Testing the carriageReturn and bell manipulators:"
41 << endLine << ".....";
42
43 cout << bell; // usa manipulador bell
44
45 // usa manipuladores carriageReturn e endLine
46 cout << carriageReturn << "-----" << endLine;
47 return 0;
48 } // fim de main

```

Testing the tab manipulator:

a        b        c

Testing the carriageReturn and bell manipulators:

-----.....

**Figura 15.11** Manipuladores de fluxo não parametrizados definidos pelo usuário.

| Manipulador de fluxo    | Descrição                                                                                                                                                                                                                                                                                                                                                  |
|-------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>skipws</code>     | Pula caracteres de espaço em branco em um fluxo de entrada. Essa configuração é redefinida com o manipulador de fluxo <code>noskipws</code> .                                                                                                                                                                                                              |
| <code>left</code>       | Saída alinhada à esquerda em um campo. Os caracteres de preenchimento aparecem à direita se necessário.                                                                                                                                                                                                                                                    |
| <code>right</code>      | Saída alinhada à direita em um campo. Os caracteres de preenchimento aparecem à esquerda se necessário.                                                                                                                                                                                                                                                    |
| <code>internal</code>   | Indica que o sinal de um número deve ser alinhado à esquerda em um campo e a magnitude de um número deve ser alinhada à direita nesse mesmo campo (isto é, caracteres de preenchimento aparecem entre o sinal e o número).                                                                                                                                 |
| <code>dec</code>        | Especifica se os inteiros devem ser tratados como valores decimais (base 10).                                                                                                                                                                                                                                                                              |
| <code>oct</code>        | Especifica se os inteiros devem ser tratados como valores octais (base 8).                                                                                                                                                                                                                                                                                 |
| <code>hex</code>        | Especifica se os inteiros devem ser tratados como valores hexadecimais (base 16).                                                                                                                                                                                                                                                                          |
| <code>showbase</code>   | Especifica se a base de um número deve ser enviada como saída na frente do número (um 0 inicial para octais; um 0x inicial ou 0X para hexadecimais). Essa configuração é redefinida com o manipulador de fluxo <code>noshowbase</code> .                                                                                                                   |
| <code>showpoint</code>  | Especifica que números de ponto flutuante devem ser enviados para a saída com um ponto de fração decimal. Normalmente, isso é utilizado com <code>fixed</code> para garantir certo número de dígitos à direita do ponto de fração decimal, mesmo se eles forem zeros. Essa configuração é redefinida com o manipulador de fluxo <code>noshowpoint</code> . |
| <code>uppercase</code>  | Especifica que as letras maiúsculas (isto é, X e de A a F) devem ser utilizadas em um inteiro hexadecimal e que a letra maiúscula E deve ser utilizada ao representar um valor de ponto flutuante na notação científica. Essa configuração é redefinida com o manipulador de fluxo <code>nouppercase</code> .                                              |
| <code>showpos</code>    | Especifica que os números positivos devem ser precedidos por um sinal de adição (+). Essa configuração é redefinida com o manipulador de fluxo <code>noshowpos</code> .                                                                                                                                                                                    |
| <code>scientific</code> | Especifica a saída de um valor de ponto flutuante em notação científica.                                                                                                                                                                                                                                                                                   |
| <code>fixed</code>      | Especifica a saída de um valor de ponto flutuante na notação de ponto fixo com um número específico de dígitos à direita do ponto de fração decimal.                                                                                                                                                                                                       |

**Figura 15.12** Manipuladores de fluxo de estado de formato `<iostream>`.

### 15.7.1 Zeros finais e pontos de fração decimal (`showpoint`)

O manipulador de fluxo `showpoint` força a geração da saída de um número de ponto flutuante com seu ponto de fração decimal e zeros finais. Por exemplo, o valor de ponto flutuante 79.0 é impresso como 79 sem utilizar `showpoint` e impresso como 79.000000 (ou quantos zeros finais forem especificados pela precisão atual) utilizando `showpoint`. Para redefinir a configuração `showpoint`, gere saída do manipulador de fluxo `noshowpoint`. O programa da Figura 15.13 mostra como utilizar o manipulador de fluxo `showpoint` para controlar a impressão de zeros finais e pontos de fração decimal para valores de ponto flutuante. Lembre-se de que a precisão-padrão de um número de ponto flutuante é 6. Quando nem o manipulador de fluxo `fixed` nem o `scientific` é utilizado, a precisão representa o número de dígitos significativos a exibir (isto é, o número total de dígitos a exibir), não o número de dígitos a exibir depois do ponto de fração decimal.

### 15.7.2 Alinhamento (`left`, `right` e `internal`)

Os manipuladores de fluxo `left` e `right` permitem que os campos sejam alinhados à esquerda com caracteres de preenchimento alinhados à direita ou alinhados à direita com caracteres de preenchimento à esquerda, respectivamente. O caractere de preenchimento é especificado pela função-membro `fill` ou pelo manipulador de fluxo parametrizado `setfill` (que discutimos na Seção 15.7.3). A Figura 15.14 utiliza os manipuladores `setw`, `left` e `right` para alinhar os dados do tipo inteiro à esquerda e à direita em um campo.

O manipulador de fluxo `internal` indica que o sinal de um número (ou base ao utilizar o manipulador de fluxo `showbase`) deve ser alinhado à esquerda dentro de um campo, que a magnitude de um número deve ser alinhada à direita e que os espaços intermediários devem ser preenchidos com o caractere de preenchimento. A Figura 15.15 mostra o fluxo manipulador `internal` que especifica o espaçamento interno (linha 15). Observe que `showpos` força o sinal de adição a ser impresso (linha 15). Para redefinir a configuração `showpos`, gere a saída do manipulador de fluxo `noshowpos`.

```

1 // Figura 15.13: Fig15_13.cpp
2 // Utilizando showpoint para controlar a impressão de
3 // zeros finais e pontos de fração decimal para doubles.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7 using std::showpoint;
8
9 int main()
10 {
11 // exibe valores double com formato de fluxo-padrão
12 cout << "Before using showpoint" << endl
13 << "9.9900 prints as: " << 9.9900 << endl
14 << "9.9000 prints as: " << 9.9000 << endl
15 << "9.0000 prints as: " << 9.0000 << endl << endl;
16
17 // exibe valor double depois de showpoint
18 cout << showpoint
19 << "After using showpoint" << endl
20 << "9.9900 prints as: " << 9.9900 << endl
21 << "9.9000 prints as: " << 9.9000 << endl
22 << "9.0000 prints as: " << 9.0000 << endl;
23 return 0;
24 } // fim de main

```

Before using showpoint

9.9900 prints as: 9.99

9.9000 prints as: 9.9

9.0000 prints as: 9

After using showpoint

9.9900 prints as: 9.99000

9.9000 prints as: 9.90000

9.0000 prints as: 9.00000

**Figura 15.13** Controlando a impressão de zeros finais e pontos de fração decimal em valores de ponto flutuante.

```

1 // Figura 15.14: Fig15_14.cpp
2 // Demonstrando o alinhamento à esquerda e à direita.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::left;
7 using std::right;
8
9 #include <iomanip>
10 using std::setw;
11
12 int main()
13 {
14 int x = 12345;
15
16 // exibe x alinhado à direita (padrão)
17 cout << "Default is right justified:" << endl

```

**Figura 15.14** Alinhamento à esquerda e à direita com os manipuladores de fluxo left e right.

(continua)

```

18 << setw(10) << x;
19
20 // utiliza manipulador left para exibir x alinhado à esquerda
21 cout << "\n\nUse std::left to left justify x:\n"
22 << left << setw(10) << x;
23
24 // utiliza manipulador right para exibir x alinhado à direita
25 cout << "\n\nUse std::right to right justify x:\n"
26 << right << setw(10) << x << endl;
27 return 0;
28 } // fim de main

```

Default is right justified:

12345

Use std::left to left justify x:

12345

Use std::right to right justify x:

12345

**Figura 15.14** Alinhamento à esquerda e à direita com os manipuladores de fluxo `left` e `right`.

(continuação)

```

1 // Figura 15.15: Fig15_15.cpp
2 // Imprimindo um inteiro com espaçamento interno e sinal de adição.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::internal;
7 using std::showpos;
8
9 #include <iomanip>
10 using std::setw;
11
12 int main()
13 {
14 // exibe o valor com espaçamento interno e sinal de adição
15 cout << internal << showpos << setw(10) << 123 << endl;
16 return 0;
17 } // fim de main

```

+ 123

**Figura 15.15** Imprimindo um inteiro com espaçamento interno e sinal de adição.

### 15.7.3 Preenchimento (`fill`, `setfill`)

A **função-membro `fill`** especifica o caractere de preenchimento a ser utilizado com campos alinhados; se nenhum valor é especificado, os espaços são utilizados para o preenchimento. A função `fill` retorna o caractere de preenchimento anterior. O **manipulador `setfill`** também configura o caractere de preenchimento. A Figura 15.16 demonstra o uso da função-membro `fill` (linha 40) e do manipulador de fluxo `setfill` (linhas 44 e 47) para configurar o caractere de preenchimento.

```

1 // Figura 15.16: Fig15_16.cpp
2 // Utilizando a função-membro fill e o manipulador de fluxo setfill para alterar
3 // o caractere de preenchimento para campos maiores do que o valor impresso.
4 #include <iostream>
5 using std::cout;
6 using std::dec;
7 using std::endl;
8 using std::hex;
9 using std::internal;
10 using std::left;
11 using std::right;
12 using std::showbase;
13
14 #include <iomanip>
15 using std::setfill;
16 using std::setw;
17
18 int main()
19 {
20 int x = 10000;
21
22 // exibe x
23 cout << x << " printed as int right and left justified\n"
24 << "and as hex with internal justification.\n"
25 << "Using the default pad character (space):" << endl;
26
27 // exibe x com base
28 cout << showbase << setw(10) << x << endl;
29
30 // exibe x com alinhamento à esquerda
31 cout << left << setw(10) << x << endl;
32
33 // exibe x como hexadecimal com alinhamento interno
34 cout << internal << setw(10) << hex << x << endl << endl;
35
36 cout << "Using various padding characters:" << endl;
37
38 // exibe x utilizando caracteres de preenchimento (à direita)
39 cout << right;
40 cout.fill('*');
41 cout << setw(10) << dec << x << endl;
42
43 // exibe x utilizando caracteres de preenchimento (alinhamento à esquerda)
44 cout << left << setw(10) << setfill('%') << x << endl;
45
46 // exibe x utilizando caracteres de preenchimento (alinhamento interno)
47 cout << internal << setw(10) << setfill('^') << hex
48 << x << endl;
49 return 0;
50 } // fim de main

```

```

10000 printed as int right and left justified
and as hex with internal justification.
Using the default pad character (space):
10000

```

**Figura 15.16** Utilizando a função-membro fill e o manipulador de fluxo setfill para alterar o caractere de preenchimento dos campos maiores que os valores sendo impressos. (continua)

```
10000
0x 2710
```

Using various padding characters:

```
*****10000
10000%%%%%
0x~~~~~2710
```

**Figura 15.16** Utilizando a função-membro `fill` e o manipulador de fluxo `setfill` para alterar o caractere de preenchimento dos campos maiores que os valores sendo impressos. (continuação)

### 15.7.4 Base de fluxo integral (dec, oct, hex, showbase)

O C++ fornece os manipuladores de fluxo `dec`, `hex` e `oct` para especificar que os inteiros devem ser exibidos como valores decimal, hexadecimal e octal, respectivamente. As inserções de fluxo assumem o padrão de decimal se nenhum desses manipuladores for utilizado. Com a extração de fluxo, os inteiros prefixados com 0 (zero) são tratados como valores octais, os inteiros prefixados com 0x ou 0X são tratados como valores hexadecimais e todos os outros inteiros são tratados como valores decimais. Uma vez que uma base particular é especificada para um fluxo, todos os inteiros nesse fluxo são processados utilizando essa base até que uma base diferente seja especificada ou até que o programa termine.

O manipulador de fluxo `showbase` força a geração da saída da base de um valor integral. A saída dos números decimais é gerada por padrão; a dos números octais, com um 0 inicial; e a dos números hexadecimais, com 0x inicial ou 0X inicial (como discutimos na Seção 15.7.6, o manipulador de fluxo `uppercase` determina a opção que é escolhida). A Figura 15.17 demonstra o uso do manipulador de fluxo `showbase` para forçar um inteiro a imprimir nos formatos decimal, octal e hexadecimal. Para redefinir a configuração `showbase`, gere a saída do manipulador de fluxo `noshowbase`.

```
1 // Figura 15.17: Fig15_17.cpp
2 // Utilizando o manipulador de fluxo showbase.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::hex;
7 using std::oct;
8 using std::showbase;
9
10 int main()
11 {
12 int x = 100;
13
14 // utiliza showbase para mostrar a base de número
15 cout << "Printing integers preceded by their base:" << endl
16 << showbase;
17
18 cout << x << endl; // imprime o valor decimal
19 cout << oct << x << endl; // imprime o valor octal
20 cout << hex << x << endl; // imprime o valor hexadecimal
21 return 0;
22 } // fim de main
```

```
Printing integers preceded by their base:
100
0144
0x64
```

**Figura 15.17** Manipulador de fluxo `showbase`.

### 15.7.5 Números de ponto flutuante; notação científica e fixa (scientific, fixed)

Os manipuladores de fluxo `scientific` e `fixed` controlam o formato de saída de números de ponto flutuante. O manipulador de fluxo `scientific` força a exibição da saída de um número de ponto flutuante no formato científico. O manipulador de fluxo `fixed` força um número de ponto flutuante a exibir um número específico de dígitos (como especificado pela função-membro `precision` ou pelo manipulador de fluxo `setprecision`) à direita do ponto de fração decimal. Sem utilizar outro manipulador, o valor de número de ponto flutuante determina o formato de saída.

A Figura 15.18 demonstra a exibição dos números de ponto flutuante nos formatos fixo e científico utilizando os manipuladores de fluxo `scientific` (linha 21) e `fixed` (linha 25). O formato de expoente na notação científica poderia diferir entre diferentes compiladores.

### 15.7.6 Controle de letras maiúsculas/minúsculas (uppercase)

O manipulador de fluxo `uppercase` gera saída de uma letra maiúscula X ou E com valores do tipo inteiro hexadecimal ou com valores de ponto flutuante de notação científica, respectivamente (Figura 15.19). Utilizar o manipulador de fluxo `uppercase` também faz com que todas as letras em um valor hexadecimal sejam letras maiúsculas. Por padrão, as letras de valores hexadecimais e os expoentes nos valores de ponto flutuante na notação científica aparecem em letras minúsculas. Para redefinir a configuração `uppercase`, gere saída do manipulador de fluxo `nouppercase`.

```

1 // Figura 15.18: Fig15_18.cpp
2 // Exibindo valores de ponto flutuante nos formatos
3 // científico e fixo do sistema-padrão.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7 using std::fixed;
8 using std::scientific;
9
10 int main()
11 {
12 double x = 0.001234567;
13 double y = 1.946e9;
14
15 // exibe X e Y no formato-padrão
16 cout << "Displayed in default format:" << endl
17 << x << '\t' << y << endl;
18
19 // exibe X e Y no formato científico
20 cout << "\nDisplayed in scientific format:" << endl
21 << scientific << x << '\t' << y << endl;
22
23 // exibe X e Y no formato fixo
24 cout << "\nDisplayed in fixed format:" << endl
25 << fixed << x << '\t' << y << endl;
26 return 0;
27 } // fim de main

```

Displayed in default format:  
0.00123457      1.946e+009

Displayed in scientific format:  
1.234567e-003    1.946000e+009

Displayed in fixed format:  
0.001235      1946000000.000000

**Figura 15.18** Valores de ponto flutuante exibidos nos formatos-padrão, científico e fixo.



```

1 // Figura 15.19: Fig15_19.cpp
2 // Manipulador de fluxo uppercase.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::hex;
7 using std::showbase;
8 using std::uppercase;
9
10 int main()
11 {
12 cout << "Printing uppercase letters in scientific" << endl
13 << "notation exponents and hexadecimal values:" << endl;
14
15 // utiliza std::uppercase para exibir letras maiúsculas; utiliza std::hex e
16 // std::showbase para exibir o valor hexadecimal e sua base
17 cout << uppercase << 4.345e10 << endl
18 << hex << showbase << 123456789 << endl;
19 return 0;
20 } // fim de main

```

```

Printing uppercase letters in scientific
notation exponents and hexadecimal values:
4.345E+010
0X75BCD15

```

**Figura 15.19** Manipulador de fluxo uppercase.

### 15.7.7 Especificando o formato booleano (boolalpha)

O C++ fornece o tipo de dados `bool`, cujos valores podem ser `false` ou `true`, como uma alternativa preferida ao estilo antigo de utilizar 0 para indicar `false` e não-zero para indicar `true`. Uma variável `bool` é enviada para a saída como 0 ou 1 por padrão. Entretanto, podemos utilizar o manipulador de fluxo `boolalpha` a fim de configurar o fluxo de saída para exibir valores `bool` como as strings ‘true’ e ‘false’. Utilize o manipulador de fluxo `noboolalpha` para configurar o fluxo de saída para exibir valores `bool` como inteiros (isto é, a configuração-padrão). O programa da Figura 15.20 demonstra esses manipuladores de fluxo. A linha 14 exibe o valor `bool`, o qual a linha 11 configura como `true`, como um inteiro. A linha 18 utiliza o manipulador `boolalpha` para exibir o valor `bool` como uma string. As linhas 21–22 então alteram o valor de `bool` e utilizam o manipulador `noboolalpha`, portanto a linha 25 pode exibir o valor `bool` como um inteiro. A linha 29 utiliza o manipulador `boolalpha` para exibir o valor `bool` como uma string. Tanto `boolalpha` como `noboolalpha` são configurações ‘aderentes’.



#### Boa prática de programação 15.1

*Exibir valores `bool` como `true` ou `false`, em vez de não-zero ou 0, respectivamente, torna as saídas de programa mais claras.*

```

1 // Figura 15.20: Fig15_20.cpp
2 // Demonstrando manipuladores de fluxo boolalpha e noboolalpha.
3 #include <iostream>
4 using std::boolalpha;
5 using std::cout;
6 using std::endl;
7 using std::noboolalpha;
8
9 int main()

```

**Figura 15.20** Manipuladores de fluxo `boolalpha` e `noboolalpha`.

(continua)

```

10 {
11 bool booleanValue = true;
12
13 // exibe booleanValue true padrão
14 cout << "booleanValue is " << booleanValue << endl;
15
16 // exibe booleanValue depois de utilizar boolalpha
17 cout << "booleanValue (after using boolalpha) is "
18 << boolalpha << booleanValue << endl << endl;
19
20 cout << "switch booleanValue and use noboolalpha" << endl;
21 booleanValue = false; // altera booleanValue
22 cout << noboolalpha << endl; // utiliza noboolalpha
23
24 // exibe booleanValue false padrão depois de utilizar noboolalpha
25 cout << "booleanValue is " << booleanValue << endl;
26
27 // exibe booleanValue depois de utilizar boolalpha novamente
28 cout << "booleanValue (after using boolalpha) is "
29 << boolalpha << booleanValue << endl;
30 return 0;
31 } // fim de main

```

```

booleanValue is 1
booleanValue (after using boolalpha) is true

switch booleanValue and use noboolalpha

booleanValue is 0
booleanValue (after using boolalpha) is false

```

**Figura 15.20** Manipuladores de fluxo boolalpha e noboolalpha.

(continuação)

### 15.7.8 Configurando e redefinindo o estado de formato via função-membro flags

Por toda a Seção 15.7, utilizamos os manipuladores de fluxo para alterar as características de formatos de saída. Agora discutimos como restaurar um formato do fluxo de saída ao seu estado-padrão depois de aplicar várias manipulações. A função-membro **flags** sem um argumento retorna as configurações de formato atuais como um tipo de dados **fmtflags** (da classe **ios\_base**), que representa o **estado de formato**. A função-membro **flags** com um argumento **fmtflags** configura o estado de formato tal como especificado pelo argumento e retorna as configurações de estado anteriores. As configurações iniciais do valor que **flags** retorna podem diferir por meio de vários sistemas. O programa da Figura 15.21 utiliza a função-membro **flags** para salvar o estado de formato original do fluxo (linha 22) e, então, restaurar as configurações do formato original (linha 30).

```

1 // Figura 15.21: Fig15_21.cpp
2 // Demonstrando a função-membro flags.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::ios_base;
7 using std::oct;
8 using std::scientific;
9 using std::showbase;
10

```

**Figura 15.21** Função-membro flags.

(continua)

```

11 int main()
12 {
13 int integerValue = 1000;
14 double doubleValue = 0.0947628;
15
16 // exibe os valores flags, int e double (formato original)
17 cout << "The value of the flags variable is: " << cout.flags()
18 << "\nPrint int and double in original format:\n"
19 << integerValue << '\t' << doubleValue << endl << endl;
20
21 // utiliza a função flags cout para salvar o formato original
22 ios_base::fmtflags originalFormat = cout.flags();
23 cout << showbase << oct << scientific; // altera o formato
24
25 // exibe os valores flags, int e double (novo formato)
26 cout << "The value of the flags variable is: " << cout.flags()
27 << "\nPrint int and double in a new format:\n"
28 << integerValue << '\t' << doubleValue << endl << endl;
29
30 cout.flags(originalFormat); // restaura formato
31
32 // exibe os valores flags, int e double (formato original)
33 cout << "The restored value of the flags variable is: "
34 << cout.flags()
35 << "\nPrint values in original format again:\n"
36 << integerValue << '\t' << doubleValue << endl;
37 return 0;
38 } // fim de main

```

```

The value of the flags variable is: 513
Print int and double in original format:
1000 0.0947628

```

```

The value of the flags variable is: 012011
Print int and double in a new format:
01750 9.476280e-002

```

```

The restored value of the flags variable is: 513
Print values in original format again:
1000 0.0947628

```

Figura 15.21 Função-membro flags.

(continuação)

## 15.8 Estados de erro de fluxo

O estado de um fluxo pode ser testado por bits na classe `ios_base`. Em breve, mostraremos como testar esses bits, no exemplo da Figura 15.22.

O `eofbit` é configurado para um fluxo de entrada depois que o fim do arquivo é encontrado. Um programa pode utilizar a função-membro `eof` para determinar se o fim do arquivo foi encontrado em um fluxo depois de uma tentativa de extrair dados além do fim do fluxo. A chamada

```
cin.eof()
```

retorna `true` se o fim do arquivo foi encontrado em `cin`, e `false` caso contrário.

O `failbit` é configurado para um fluxo quando ocorre um erro de formato no fluxo, como quando o programa está inserindo inteiros e um caractere não-dígito é encontrado no fluxo de entrada. Quando um erro assim ocorre, os caracteres não são perdidos. A função-membro `fail` relata que uma operação de fluxo falhou; normalmente, é possível se recuperar desses erros.

O `badbit` é configurado para um fluxo quando ocorre um erro que resulta na perda de dados. A função-membro `bad` informa se uma operação de fluxo falhou. Em geral, não é possível se recuperar dessas falhas.

O **goodbit** é configurado para um fluxo se nenhum dos bits eofbit, failbit ou badbit estiver configurado para o fluxo.

A função-membro **good** retorna true se as funções bad, fail e eof retornarem false. As operações de E/S devem ser realizadas somente em fluxos 'válidos'.

A função-membro **rdstate** retorna o estado de erro do fluxo. Uma chamada a `cout.rdstate`, por exemplo, retornaria o estado do fluxo, que poderia então ser testado por uma instrução switch que examina eofbit, badbit, failbit e goodbit. O meio preferido de testar o estado de um fluxo é utilizar as funções-membro eof, bad, fail e good — utilizá-las não requer que o programador conheça bits de status particulares.

A função-membro **clear** é utilizada para restaurar o estado de um fluxo a um estado 'válido', para que a E/S possa prosseguir nesse fluxo. O argumento-padrão para clear é goodbit, então a instrução

```
cin.clear();
```

limpa cin e configura goodbit para o fluxo. A instrução

```
cin.clear(ios::failbit)
```

configura o failbit. O programador poderia querer fazer isso ao realizar a entrada em cin com um tipo definido pelo usuário e encontrar um problema. O nome clear pode parecer inadequado nesse contexto, mas é correto.

```
1 // Figura 15.22: Fig15_22.cpp
2 // Testando estados de erro.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6 using std::endl;
7
8 int main()
9 {
10 int integerValue;
11
12 // exibe resultados de funções cin
13 cout << "Before a bad input operation:"
14 << "\ncin.rdstate(): " << cin.rdstate()
15 << "\n cin.eof(): " << cin.eof()
16 << "\n cin.fail(): " << cin.fail()
17 << "\n cin.bad(): " << cin.bad()
18 << "\n cin.good(): " << cin.good()
19 << "\n\nExpects an integer, but enter a character: ";
20
21 cin >> integerValue; // insere valor de caractere
22 cout << endl;
23
24 // exibe resultados de funções cin depois de entrada inválida
25 cout << "After a bad input operation:"
26 << "\ncin.rdstate(): " << cin.rdstate()
27 << "\n cin.eof(): " << cin.eof()
28 << "\n cin.fail(): " << cin.fail()
29 << "\n cin.bad(): " << cin.bad()
30 << "\n cin.good(): " << cin.good() << endl << endl;
31
32 cin.clear(); // limpa o fluxo
33
34 // exibe resultados de funções cin depois de limpar cin
35 cout << "After cin.clear()" << "\ncin.fail(): " << cin.fail()
36 << "\ncin.good(): " << cin.good() << endl;
37 return 0;
38 } // fim de main
```

Figura 15.22 Testando estados de erro.

(continua)

```

Before a bad input operation:
cin.rdstate(): 0
cin.eof(): 0
cin.fail(): 0
cin.bad(): 0
cin.good(): 1

Expects an integer, but enter a character: A

After a bad input operation:
cin.rdstate(): 2
cin.eof(): 0
cin.fail(): 1
cin.bad(): 0
cin.good(): 0

After cin.clear()
cin.fail(): 0
cin.good(): 1

```

**Figura 15.22** Testando estados de erro.

(continuação)

O programa da Figura 15.22 demonstra as funções-membro `rdstate`, `eof`, `fail`, `bad`, `good` e `clear`. [Nota: A saída de valores reais pode diferir nos compiladores diferentes.]

A função-membro `operator!` de `basic_istream` retorna `true` se o `badbit` estiver configurado, se o `failbit` estiver configurado ou se ambos estiverem configurados. A função-membro `operator void *` retorna `false` (0) se o `badbit` estiver configurado, se `failbit` estiver configurado ou se ambos estiverem configurados. Essas funções são úteis no processamento de arquivo quando uma condição `true/false` estiver em teste sob o controle de uma instrução de seleção ou de repetição.

## 15.9 Associando um fluxo de saída a um fluxo de entrada

Os aplicativos interativos geralmente envolvem um `istream` para a entrada e um `ostream` para a saída. Quando uma mensagem de solicitação aparece na tela, o usuário responde inserindo os dados apropriados. Obviamente, a solicitação precisa aparecer antes de a operação de entrada continuar. Com o buffer de saída, as saídas só aparecem quando o buffer é preenchido, quando as saídas são esvaziadas explicitamente pelo programa ou automaticamente no fim do programa. O C++ fornece a função-membro `tie` para sincronizar (isto é, ‘associar’) a operação de um `istream` e um `ostream` para assegurar que as saídas aparecem antes de suas entradas subseqüentes. A chamada

```
cin.tie(&cout);
```

associa `cout` (um `ostream`) a `cin` (um `istream`). De fato, essa chamada particular é redundante, porque o C++ realiza automaticamente essa operação para criar o ambiente-padrão de entrada/saída de um usuário. Entretanto, o usuário associaria outros pares `istream/ostream` explicitamente. Para desassociar um fluxo de entrada, `istream`, de um fluxo de saída, utilize a chamada

```
istream.tie(0);
```

## 15.10 Síntese

Este capítulo resumiu como o C++ realiza a entrada/saída utilizando fluxos. Você aprendeu sobre as classes e objetos de E/S de fluxo, bem como sobre a hierarquia de classes de template de E/S de fluxo. Discutimos as capacidades de saída formatada e não formatada do `ostream` realizadas pelas funções `put` e `write`. Você viu exemplos de utilização de capacidades de entrada formatada e não formatada do `istream` pelas funções `eof`, `get`, `getline`, `peek`, `putback`, `ignore` e `read`. Em seguida, discutimos os manipuladores de fluxo e as funções-membro que realizam tarefas de formatação — `dec`, `oct`, `hex` e `setbase` para exibir inteiros; `precision` e `setprecision` para controlar precisão de ponto flutuante; e `width` e `setw` para configurar largura de campo. Você também aprendeu sobre os manipuladores de formatação `istream` e as funções-membro adicionais — `showpoint` para exibir ponto de fração decimal e zeros finais; `left`, `right` e `internal` para alinhamento; `fill` e `setfill` para preenchimento; `scientific` e `fixed` para exibir números de ponto flutuante na notação científica e na notação fixa; `uppercase` para controlar letras maiúsculas/minúsculas; `boolalpha` para especificar o formato booleano; e `flags` e `fmtflags` para redefinir o estado de formato.

No próximo capítulo, introduzimos o tratamento de exceções, que permite aos programadores tratar certos problemas que podem ocorrer durante a execução de um programa. Demonstramos as técnicas básicas de tratamento de exceções que freqüentemente permitem a um programa continuar executando como se nenhum problema tivesse sido encontrado. Apresentamos também as várias classes de tratamento de exceções oferecidas pela C++ Standard Library.

## Resumo

- As operações de E/S são realizadas de maneira sensível ao tipo dos dados.
- A E/S do C++ ocorre em fluxos. Um fluxo é uma seqüência de bytes.
- Os mecanismos de E/S do sistema movem os bytes a partir de dispositivos para a memória e vice-versa de maneira eficiente e confiável.
- O C++ fornece capacidades de E/S de ‘baixo nível’ e ‘alto nível’. As capacidades de E/S de baixo nível especificam que um número de bytes deve ser transferido de dispositivos para a memória ou da memória para os dispositivos. A E/S de alto nível é realizada com bytes agrupados em unidades significativas tais como inteiros, números de ponto flutuante, caracteres, strings e tipos definidos pelo usuário.
- O C++ fornece tanto operações de E/S não formatadas como formatadas. As transferências de E/S não formatadas são rápidas, mas processam dados brutos que são difíceis de utilizar. A E/S formatada processa dados em unidades significativas, mas requer tempo extra de processamento que pode degradar o desempenho de transferências de dados de alto volume.
- O arquivo de cabeçalho `<iostream>` declara todas as operações de E/S de fluxo.
- O cabeçalho `<iomanip>` declara os manipuladores parametrizados de fluxo.
- O cabeçalho `<fstream>` declara as operações de processamento de arquivo.
- O template `basic_istream` suporta as operações de entrada de fluxo.
- O template `basic_ostream` suporta operações de saída de fluxo.
- O template `basic_istream` suporta tanto operações de entrada como de saída de fluxo.
- Os templates `basic_istream` e `basic_ostream` são derivados por meio da herança simples do template `basic_ios`.
- O template `basic_istream` é derivado por meio da herança múltipla tanto do template `basic_istream` como do `basic_ostream`.
- O operador de deslocamento para a esquerda (`<<`) é sobrecarregado para designar a saída de fluxo e é referido como operador de inserção de fluxo.
- O operador de deslocamento para a direita (`>>`) é sobrecarregado para designar a entrada de fluxo e é referido como operador de extração de fluxo.
- O objeto `istream cin` é associado ao dispositivo de entrada-padrão, normalmente o teclado.
- O objeto `ostream cout` é associado ao dispositivo de saída-padrão, normalmente a tela.
- O objeto `ostream cerr` é associado ao dispositivo de erro-padrão. As saídas para `cerr` não são armazenadas em buffer; cada inserção para `cerr` aparece imediatamente.
- O compilador C++ determina os tipos de dados automaticamente para a entrada e para a saída.
- Os endereços são exibidos em formato hexadecimal por padrão.
- Para imprimir o endereço em uma variável de ponteiro, faça coerção do ponteiro para `void *`.
- A função-membro `put` gera saída de um caractere. As chamadas para `put` podem ser em cascata.
- A entrada de fluxo é realizada com o operador de extração de fluxo `>>`. Esse operador pula automaticamente os caracteres de espaço em branco no fluxo de entrada.
- O operador `>>` retorna `false` depois que o fim do arquivo é encontrado em um fluxo.
- A extração de fluxo faz com que `failbit` seja configurado para uma entrada inadequada e que `badbit` seja configurado se a operação falhar.
- Uma série de valores pode ser inserida utilizando a operação de extração de fluxo em um cabeçalho de loop `while`. A extração retorna 0 quando o fim do arquivo é encontrado.
- A função-membro `get` sem argumentos insere um caractere e o retorna; o EOF é retornado se o fim do arquivo for encontrado no fluxo.
- A função-membro `get` com um argumento de referência de caractere insere o próximo caractere a partir do fluxo de entrada e o armazena no argumento de caractere. Essa versão de `get` retorna uma referência ao objeto `istream` para o qual a função-membro `get` está sendo invocada.
- A função-membro `get` com três argumentos — um array de caracteres, um limite de tamanho e um delimitador (com o valor-padrão de nova linha) — lê os caracteres do fluxo de entrada até um máximo de limite – 1 caractere e termina, ou termina quando o delimitador é lido. A string de entrada é terminada com um caractere nulo. O delimitador não é colocado no array de caracteres, mas permanece no fluxo de entrada.

- A função-membro `getline` opera de modo semelhante à função-membro `get` de três argumentos. A função `getline` remove o delimitador do fluxo de entrada, mas não o armazena na string.
- A função-membro `ignore` pula o número especificado de caracteres (o padrão é 1) no fluxo de entrada; ela termina se o delimitador especificado for encontrado (o delimitador-padrão é EOF).
- A função-membro `putback` coloca o caractere anterior obtido por um `get` em um fluxo de volta nesse fluxo.
- A função-membro `peek` retorna o próximo caractere de um fluxo de entrada, mas não extrai (remove) o caractere do fluxo.
- O C++ oferece E/S fortemente tipada (*type-safe*). Se dados inesperados são processados pelos operadores `<<` e `>>`, vários bits de erro são configurados, os quais o usuário pode testar para determinar se uma operação de E/S foi ou não bem-sucedida. Se o operador `<<` não tiver sido sobrecarregado para um tipo definido pelo usuário, um erro de compilador é informado.
- A E/S não formatada é realizada com as funções-membro `read` e `write`. Essas realizam a entrada ou saída de um número de bytes para ou a partir da memória, começando em um endereço de memória designado. Na entrada ou saída desses bytes, eles são tratados como bytes brutos sem formatação.
- A função-membro `gcount` retorna o número de caracteres inseridos pela operação `read` anterior nesse fluxo.
- A função-membro `read` insere um número especificado de caracteres em um array de caracteres. `failbit` é configurado se um número de caracteres menor que o especificado for lido.
- Para alterar a base dos inteiros que são enviados para a saída, utilize o manipulador `hex` para configurar a base como hexadecimal (base 16) ou `oct` para configurá-la como octal (base 8). Utilize o manipulador `dec` para redefinir a base como decimal. A base permanece a mesma até ser alterada explicitamente.
- O manipulador de fluxo parametrizado `setbase` também configura a base para a saída de inteiro. `setbase` aceita um argumento de inteiro de 10, 8 ou 16 para configurar a base.
- A precisão de ponto flutuante pode ser controlada com o manipulador de fluxo `setprecision` ou com a função-membro `precision`. Ambos configuram a precisão para todas as operações de saída subsequentes até a próxima chamada de configuração de precisão. A função-membro `precision` sem argumento retorna o valor de precisão atual.
- Os manipuladores parametrizados requerem a inclusão do arquivo de cabeçalho `<iomanip>`.
- A função-membro `width` configura a largura de campo e retorna a largura anterior. Os valores mais estreitos do que o campo são preenchidos com caracteres de preenchimento. A configuração de largura de campo se aplica somente à próxima inserção ou extração; a largura de campo é configurada implicitamente como 0 (a saída de valores subsequentes terá a largura necessária). Os valores mais largos do que um campo são impressos em sua totalidade. A função `width` sem argumento retorna a configuração de largura atual. O manipulador `setw` também configura a largura.
- Para a entrada, o manipulador de fluxo `setw` estabelece o tamanho máximo de uma string; se uma string maior é inserida, a linha maior é dividida em partes não maiores do que as do tamanho designado.
- Os programadores podem criar seus próprios manipuladores de fluxo.
- O manipulador de fluxo `showpoint` força a geração da saída de um número de ponto flutuante com um ponto de fração decimal e com o número de dígitos significativos especificado pela precisão.
- Os manipuladores de fluxo `left` e `right` fazem com que os campos sejam alinhados à esquerda com caracteres de preenchimento à direita ou alinhados à direita com caracteres de preenchimento à esquerda.
- O manipulador de fluxo `internal` indica que o sinal de um número (ou a base ao utilizar manipulador de fluxo `showbase`) deve ser alinhado à esquerda dentro de um campo, sua magnitude deve ser alinhada à direita e os espaços intermediários devem ser preenchidos com o caractere de preenchimento.
- A função-membro `fill` especifica que o caractere de preenchimento seja utilizado com os manipuladores de fluxo `left`, `right` e `internal` (espaço é o padrão); o caractere de preenchimento anterior é retornado. O manipulador de fluxo `setfill` também configura o caractere de preenchimento.
- Os manipuladores de fluxo `oct`, `hex` e `dec` especificam que os inteiros devem ser tratados como valores octal, hexadecimal ou decimal, respectivamente. A saída de inteiro assume o padrão de decimal se nenhum desses bits estiver configurado; as extrações de fluxo processam os dados na forma que os dados são fornecidos.
- O manipulador de fluxo `showbase` força a geração da saída da base de um valor integral.
- O manipulador de fluxo `scientific` é utilizado para gerar a saída de um número de ponto flutuante no formato científico. O manipulador de fluxo `fixed` é utilizado para gerar a saída de um número de ponto flutuante com a precisão especificada pela função-membro `precision`.
- O manipulador de fluxo `uppercase` força a geração da saída de um X ou E maiúsculo com inteiros hexadecimais ou com valores de ponto flutuante da notação científica, respectivamente. Quando configurado, `uppercase` faz com que todas as letras em um valor hexadecimal sejam maiúsculas.
- A função-membro `flags` sem argumento retorna o valor `long` das configurações de estado de formato atuais. A função `flags` com um argumento `long` configura o estado de formato especificado pelo argumento.



- O estado de um fluxo pode ser testado por bits na classe `ios_base`.
- O `eofbit` é configurado para um fluxo de entrada depois que o fim do arquivo é encontrado durante uma operação de entrada. A função-membro `eof` informa se o `eofbit` foi configurado.
- O `failbit` é configurado para um fluxo quando ocorre um erro de formato no fluxo. A função-membro `fail` informa se uma operação de fluxo falhou; normalmente, é possível se recuperar desses erros.
- O `badbit` é configurado para um fluxo quando ocorre um erro que resulta em perda de dados. A função-membro `bad` informa se tal operação de fluxo falhou. Normalmente não é possível se recuperar dessas falhas sérias.
- A função-membro `good` retorna `true` se as funções `bad`, `fail` e `eof` retornarem `false`. As operações de E/S devem ser realizadas somente em fluxos ‘válidos’.
- A função-membro `rdstate` retorna o estado de erro do fluxo.
- A função-membro `clear` restaura um fluxo ao estado ‘bom’, para que a E/S possa prosseguir nesse fluxo.
- O C++ fornece a função-membro `tie` para sincronizar as operações `istream` e `ostream` para assegurar que saídas sejam exibidas antes de entradas subsequentes.

## Terminologia

|                                                              |                                                                    |                                                                       |
|--------------------------------------------------------------|--------------------------------------------------------------------|-----------------------------------------------------------------------|
| <code>&lt;iomanip&gt;</code> , arquivo de cabeçalho          | formatação na memória                                              | operador de inserção de fluxo ( <code>&lt;&lt;</code> )               |
| 0 inicial (octal)                                            | <code>fstream</code>                                               | operator <code>void*</code> , função-membro de <code>basic_ios</code> |
| 0x inicial ou 0X (hexadecimal)                               | função-membro <code>bad</code> de <code>basic_ios</code>           | operator <code>!</code> , função-membro de <code>basic_ios</code>     |
| <code>badbit</code>                                          | <code>gcount</code> , função-membro de <code>basic_istream</code>  | <code>ostream</code>                                                  |
| <code>basic_fstream</code> , template de classe              | <code>get</code> , função-membro de <code>basic_istream</code>     | <code>peek</code> , função-membro de <code>basic_istream</code>       |
| <code>basic_ifstream</code> , template de classe             | <code>getline</code> , função-membro de <code>basic_istream</code> | precisão-padrão                                                       |
| <code>basic_ios</code> , template de classe                  | <code>good</code> , função-membro de <code>basic_ios</code>        | <code>precision</code> , função-membro de <code>ios_base</code>       |
| <code>basic_iostream</code> , template de classe             | <code>hex</code> , manipulador de fluxo                            | preenchimento                                                         |
| <code>basic_istream</code> , template de classe              | <code>ifstream</code>                                              | <code>put</code> , função-membro de <code>basic_ostream</code>        |
| <code>basic_ofstream</code> , template de classe             | <code>ignore</code> , função-membro de <code>basic_istream</code>  | <code>putback</code> , função-membro de <code>basic_istream</code>    |
| <code>basic_ostream</code> , template de classe              | <code>internal</code> , manipulador de fluxo                       | <code>rdstate</code> , função-membro de <code>basic_ios</code>        |
| <code>boolalpha</code> , manipulador de fluxo                | <code>ios_base</code> , classe                                     | <code>read</code> , função-membro de <code>basic_istream</code>       |
| buffer de saída                                              | <code>iostream</code>                                              | <code>right</code> , manipulador de fluxo                             |
| caractere de preenchimento                                   | <code>istream</code>                                               | saída de fluxo                                                        |
| caractere de preenchimento-padrão (espaço)                   | largura de campo                                                   | saída não armazenada em buffer                                        |
| <code>clear</code> , função-membro de <code>basic_ios</code> | <code>left</code> , manipulador de fluxo                           | <code>scientific</code> , manipulador de fluxo                        |
| <code>dec</code> , manipulador de fluxo                      | manipulador de fluxo                                               | <code>setbase</code> , manipulador de fluxo                           |
| E/S formatada                                                | manipulador de fluxo <code>fixed</code>                            | <code>setfill</code> , manipulador de fluxo                           |
| E/S fortemente tipada (type-safe)                            | manipulador de fluxo parametrizado                                 | <code>setprecision</code> , manipulador de fluxo                      |
| E/S não formatada                                            | <code>noboolalpha</code> , manipulador de fluxo                    | <code>setw</code> , manipulador de fluxo                              |
| entrada de fluxo                                             | <code>noshowbase</code> , manipulador de fluxo                     | <code>showbase</code> , manipulador de fluxo                          |
| <code>eof</code> , função-membro de <code>basic_ios</code>   | <code>noshowpoint</code> , manipulador de fluxo                    | <code>showpoint</code> , manipulador de fluxo                         |
| <code>eofbit</code>                                          | <code>noshowpos</code> , manipulador de fluxo                      | <code>showpos</code> , manipulador de fluxo                           |
| estados de formato                                           | <code>noskipws</code> , manipulador de fluxo                       | <code>skipws</code> , manipulador de fluxo                            |
| <code>fail</code> , função-membro de <code>basic_ios</code>  | <code>nouppercase</code> , manipulador de fluxo                    | <code>tie</code> , função-membro de <code>basic_ios</code>            |
| <code>failbit</code>                                         | <code>oct</code> , manipulador de fluxo                            | <code>typedef</code>                                                  |
| <code>fill</code> , função-membro de <code>basic_ios</code>  | <code>ofstream</code>                                              | <code>uppercase</code> , manipulador de fluxo                         |
| fim do arquivo                                               | operador de extração de fluxo ( <code>&gt;&gt;</code> )            | <code>width</code> , manipulador de fluxo                             |
| flags, função-membro de <code>ios_base</code>                |                                                                    | <code>write</code> , função-membro de <code>basic_ostream</code>      |
| fluxos predefinidos                                          |                                                                    |                                                                       |
| <code>fmtflags</code>                                        |                                                                    |                                                                       |

## Exercícios de revisão

**15.1** Complete cada uma das seguintes sentenças:

- A entrada/saída em C++ ocorre como \_\_\_\_\_ de bytes.
- Os manipuladores de fluxo que formatam o alinhamento são \_\_\_\_\_, \_\_\_\_\_ e \_\_\_\_\_.
- A função-membro \_\_\_\_\_ pode ser utilizada para configurar e redefinir o estado de formato.

- d) A maioria dos programas C++ que fazem E/S deve incluir o arquivo de cabeçalho \_\_\_\_\_ que contém as declarações requeridas para todas as operações de fluxo de E/S.
- e) Ao utilizar manipuladores parametrizados, o arquivo de cabeçalho \_\_\_\_\_ deve ser incluído.
- f) O arquivo de cabeçalho \_\_\_\_\_ contém as declarações requeridas para processamento de arquivo de controle do usuário.
- g) A função-membro `ostream` \_\_\_\_\_ é utilizada para realizar saída não formatada.
- h) As operações de entrada são suportadas pela classe \_\_\_\_\_.
- i) As saídas para o fluxo de erro-padrão são direcionadas para o objeto fluxo \_\_\_\_\_ ou \_\_\_\_\_.
- j) As operações de saída são suportadas pela classe \_\_\_\_\_.
- k) O símbolo do operador de inserção de fluxo é \_\_\_\_\_.
- l) Os quatro objetos que correspondem aos dispositivos-padrão no sistema incluem \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_ e \_\_\_\_\_.
- m) O símbolo do operador de extração de fluxo é \_\_\_\_\_.
- n) Os manipuladores de fluxo \_\_\_\_\_, \_\_\_\_\_ e \_\_\_\_\_ especificam que inteiros devem ser exibidos nos formatos octal, hexadecimal e decimal, respectivamente.
- o) Quando utilizado, o manipulador de fluxo \_\_\_\_\_ faz com que números positivos sejam exibidos com um sinal de adição.

### 15.2 Determine se as seguintes sentenças são *verdadeiras* ou *falsas*. Se a resposta for *falsa*, explicar por quê.

- a) A função-membro de fluxo `flags` com um argumento `long` configura a variável de estado `flags` como seu argumento e retorna seu valor anterior.
- b) O operador de inserção de fluxo `<<` e o operador de extração de fluxo `>>` são sobrecarregados para tratar todos os tipos de dados-padrão — incluindo strings e endereços de memória (inserção de fluxo somente) — e todos os tipos de dados definidos pelo usuário.
- c) A função-membro de fluxo `flags` sem argumentos reinicializa o estado de formato do fluxo.
- d) O operador de extração de fluxo `>>` pode ser sobrecarregado com uma função operadora que aceita uma referência `istream` e uma referência a um tipo definido pelo usuário como argumentos e retorna uma referência `istream`.
- e) O operador de inserção de fluxo `<<` pode ser sobrecarregado com uma função operadora que aceita uma referência `istream` e uma referência a um tipo definido pelo usuário como argumentos e retorna uma referência `istream`.
- f) A entrada com o operador de extração de fluxo `>>` sempre pula caracteres iniciais de espaço em branco no fluxo de entrada, por padrão.
- g) A função-membro de fluxo `rdstate` retorna o estado atual do fluxo.
- h) O fluxo `cout` normalmente é conectado à tela de vídeo.
- i) A função-membro de fluxo `good` retorna `true` se as funções-membro `bad`, `fail` e `eof` retornarem `false`.
- j) O fluxo `cin` normalmente é conectado à tela de vídeo.
- k) Se ocorrer um erro irreversível durante uma operação de fluxo, a função-membro `bad` retornará `true`.
- l) A saída para `cerr` não é armazenada no buffer e a saída para `clog` é armazenada em buffer.
- m) O manipulador de fluxo `showpoint` força a impressão de valores de ponto flutuante com o padrão de seis dígitos de precisão, a menos que o valor de precisão tenha sido alterado, caso em que os valores de ponto flutuante são impressos com a precisão especificada.
- n) A função-membro `ostream::put` gera saída do número especificado de caracteres.
- o) Os manipuladores de fluxo `dec`, `oct` e `hex` afetam somente a próxima operação de saída de inteiro.
- p) Por padrão, os endereços de memória são exibidos como inteiros `long`.

### 15.3 Para cada uma das seguintes sentenças, escreva uma única instrução que realiza a tarefa indicada.

- a) Gere saída da string "Enter your name: ".
- b) Utilize um manipulador de fluxo que faz com que o expoente na notação científica e as letras em valores hexadecimais sejam impressos em letras maiúsculas.
- c) Gere saída do endereço da variável `myString` do tipo `char *`.
- d) Utilize um manipulador de fluxo para assegurar que a impressão de valores de ponto flutuante seja feita em notação científica.
- e) Gere saída do endereço na variável `integerPtr` do tipo `int *`.
- f) Utilize um manipulador de fluxo de modo que, quando a saída dos valores de inteiro for gerada, a base de inteiro para os valores octal e hexadecimal seja exibida.
- g) Gere saída do valor apontado pelo `floatPtr` do tipo `float *`.
- h) Utilize uma função-membro de fluxo para configurar o caractere de preenchimento como `'*'` para imprimir em larguras de campo maiores que os valores cuja saída está sendo gerada. Escreva uma instrução separada para fazer isso com um manipulador de fluxo.
- i) Gere a saída dos caracteres `'O'` e `'K'` em uma instrução com a função `ostream::put`.
- j) Obtenha o valor do próximo caractere no fluxo de entrada sem extraí-lo do fluxo.
- k) Insira um único caractere na variável `charValue` do tipo `char`, utilizando a função-membro `istream::get` de duas maneiras diferentes.
- l) Insira e descarte os próximos seis caracteres do fluxo de entrada.
- m) Utilize a função-membro `istream::read` para inserir 50 caracteres no array `char line`.
- n) Leia 10 caracteres no array de caracteres `name`. Pare de ler os caracteres se o delimitador `'.'` for encontrado. Não remova o delimitador do fluxo de entrada. Escreva outra instrução que realiza essa tarefa e remove o delimitador da entrada.

- o) Utilize a função-membro `istream::gcount` para determinar o número de caracteres inseridos no array de caracteres `line` pela última chamada à função-membro `istream::read`, e gere saída desse número de caracteres, utilizando a função-membro `ostream::write`.
- p) Gere saída dos seguintes valores: 124, 18.376, 'Z', 1000000 e "String".
- q) Imprima a configuração de precisão atual, utilizando uma função-membro de objeto `cout`.
- r) Insira um valor de inteiro na variável `int months` e um valor de ponto flutuante na variável `float percentageRate`.
- s) Imprima 1.92, 1.925 e 1.9258 separados por tabulações e com 3 dígitos de precisão, utilizando um manipulador.
- t) Imprima o inteiro 100 em octal, hexadecimal e decimal, utilizando manipuladores de fluxo.
- u) Imprima o inteiro 100 em decimal, octal e hexadecimal, utilizando um manipulador de fluxo para alterar a base.
- v) Imprima 1234 alinhado à direita em um campo de 10 dígitos.
- w) Leia os caracteres no array de caracteres `line` até o caractere `character 'z'` ser encontrado, até um limite de 20 caracteres (incluindo um caractere de terminação nulo). Não extraia o caractere delimitador do fluxo.
- x) Utilize as variáveis de inteiro `x` e `y` para especificar a largura de campo e a precisão utilizadas para exibir o valor `double 87.4573` e exibir o valor.

**15.4** Identifique o erro em cada uma das seguintes instruções e explique como corrigi-lo.

- a) `cout << "Value of x <= y is: " << x <= y;`
- b) A seguinte instrução deve imprimir o valor de inteiro de 'c'.  
`cout << 'c';`
- c) `cout << "A string in quotes";`

**15.5** Para cada uma das seguintes instruções, mostre a saída.

- a) `cout << "12345" << endl;`  
`cout.width( 5 );`  
`cout.fill( '*' );`  
`cout << 123 << endl << 123;`
- b) `cout << setw( 10 ) << setfill( '$' ) << 10000;`
- c) `cout << setw( 8 ) << setprecision( 3 ) << 1024.987654;`
- d) `cout << showbase << oct << 99 << endl << hex << 99;`
- e) `cout << 100000 << endl << showpos << 100000;`
- f) `cout << setw( 10 ) << setprecision( 2 ) << scientific << 444.93738;`

## Respostas dos exercícios de revisão

**15.1** a) fluxos. b) `left, right, internal`. c) `flags`. d) `<iostream>`. e) `<iomanip>`. f) `<fstream>`. g) `write`. h) `istream`. i) `cerr, clog`. j) `ostream`. k) `<<`. l) `cin, cout, cerr, clog`. m) `>>`. n) `oct, hex, dec`. o) `showpos`.

**15.2** a) Falsa. A função-membro de fluxo `flags` com um `fmtflags` o argumento configura a variável de estado `flags` para seu argumento e retorna as configurações de estado anteriores. b) Falsa. Os operadores de inserção e extração de fluxo não são sobrecarregados para todos os tipos definidos pelo usuário. O programador de uma classe deve fornecer especificamente as funções de operador sobrecarregado para sobrecarregar os operadores de fluxo a utilizar com cada tipo definido pelo usuário. c) Falsa. A função-membro de fluxo `flags` sem argumentos retorna as configurações de formato atuais como um tipo de dados `fmtflags`, que representa o estado de formato. d) Verdadeira. e) Falsa. Para sobrecarregar o operador de inserção de fluxo `<<`, a função operadora sobrecarregada deve aceitar uma referência `ostream` e uma referência a um tipo definido pelo usuário como argumentos e retornar uma referência `ostream`. f) Verdadeira. g) Verdadeira. h) Verdadeira. i) Verdadeira. j) Falsa. O fluxo `cin` é conectado à entrada-padrão do computador, que normalmente é o teclado. k) Verdadeira. l) Verdadeira. m) Verdadeira. n) Falsa. A função-membro `ostream::put` gera saída de seu argumento de um único caractere. o) Falsa. Os manipuladores de fluxo `dec`, `oct` e `hex` configuram o estado de formatado da saída dos inteiros como a base especificada até que a base seja novamente alterada ou até que o programa termine. p) Falsa. Os endereços de memória são exibidos em formato hexadecimal por padrão. Para exibir endereços como inteiros `long`, o endereço deve sofrer coerção para um valor `long`.

**15.3** a) `cout << "Enter your name: ";`  
b) `cout << uppercase;`  
c) `cout << static_cast< void * >( myString );`  
d) `cout << scientific;`  
e) `cout << integerPtr;`  
f) `cout << showbase;`  
g) `cout << *floatPtr;`  
h) `cout.fill( '*' );`  
`cout << setfill( '*' );`  
i) `cout.put( 'O' ).put( 'K' );`

```

j) cin.peek();
k) charValue = cin.get();
 cin.get(charValue);
l) cin.ignore(6);
m) cin.read(line, 50);
n) cin.get(name, 10, '.');
 cin.getline(name, 10, '.');
o) cout.write(line, cin.gcount());
p) cout << 124 << ' ' << 18.376 << ' ' << "Z " << 1000000 << " String";
q) cout << cout.precision();
r) cin >> months >> percentageRate;
s) cout << setprecision(3) << 1.92 << '\t' << 1.925 << '\t' << 1.9258;
t) cout << oct << 100 << '\t' << hex << 100 << '\t' << dec << 100;
u) cout << 100 << '\t' << setbase(8) << 100 << '\t' << setbase(16) << 100;
v) cout << setw(10) << 1234;
w) cin.get(line, 20, 'z');
x) cout << setw(x) << setprecision(y) << 87.4573;

```

**15.4** a) Erro: A precedência do operador << é mais alta do que a do <=, que faz com que a instrução seja avaliada inadequadamente e também produz um erro de compilador.

Correção: Para corrigir a instrução, coloque a expressão `x <= y` entre parênteses. Esse problema ocorrerá com qualquer expressão que utiliza operadores de precedência mais baixa que a do operador << se a expressão não for colocada entre parênteses.

b) Erro: Em C++, os caracteres não são tratados como inteiros do tipo `small`, como o são em C.

Correção: Para imprimir o valor numérico para um caractere no conjunto de caracteres do computador, o caractere deve sofrer coerção para um valor inteiro, como a seguir:

```
cout << static_cast< int >('c');
```

c) Erro: Os caracteres de citação não podem ser impressos em uma string a menos que uma sequência de escape seja utilizada.

Correção: Imprima a string em uma das seguintes maneiras:

```
cout << "'" << "A string in quotes" << "'";
cout << "\"A string in quotes\"";
```

**15.5** a) 12345  
       \*\*123  
       123  
 b) \$\$\$\$10000  
 c) 1024.988  
 d) 0143  
    0x63  
 e) 100000  
    +100000  
 f) 4.45e+002

## Exercícios

**15.6** Escreva uma instrução para cada uma das seguintes sentenças:

- Imprima o inteiro 40000 alinhado à esquerda em um campo de 15 algarismos.
- Leia uma string para a variável de array de caracteres `state`.
- Imprima 200 com e sem sinal.
- Imprima o valor decimal 100 na forma de hexadecimal precedida por 0x.
- Leia os caracteres no array `charArray` até que o caractere 'p' seja encontrado, até um limite de 10 caracteres (incluindo o caractere de terminação nulo). Extraia o delimitador do fluxo de entrada e o descarte.
- Imprima 1.234 em um campo de 9 algarismos com zeros precedentes.
- Leia uma string na forma "characters" a partir da entrada-padrão. Armazene a string no array de caracteres `charArray`. Elimine as aspas do fluxo de entrada. Leia um máximo de 50 caracteres (incluindo o caractere de terminação nulo).

**15.7** Escreva um programa para testar a inserção de valores do tipo inteiro nos formatos decimal, octal e hexadecimal. Gere saída de cada inteiro lido pelo programa nos três formatos. Teste o programa com os seguintes dados de entrada: 10, 010, 0x10.

**15.8** Escreva um programa que imprime valores de ponteiro, utilizando coerções para todos os tipos de dados inteiros. Quais deles imprimem valores estranhos? Quais deles causam erros?

- 15.9** Escreva um programa para testar os resultados da impressão do valor inteiro 12345 e do valor de ponto flutuante 1.2345 em campos de vários tamanhos. O que acontece quando os valores são impressos em campos que contêm menos dígitos do que os valores?
- 15.10** Escreva um programa que imprima o valor 100.453627 arredondado para o mais próximo dígito, décimo, centésimo, milésimo e décimo de milésimo.
- 15.11** Escreva um programa que insere uma string do teclado e determina o comprimento da string. Imprima a string em um comprimento duas vezes maior que a largura do campo.
- 15.12** Escreva um programa que converte valores inteiros de temperaturas em Fahrenheit de 0 a 212 graus em valores de ponto flutuante de temperaturas em Celsius com 3 dígitos de precisão. Utilize a fórmula
- $$\text{celsius} = 5.0 / 9.0 * (\text{fahrenheit} - 32);$$
- para realizar o cálculo. A saída deve ser impressa em duas colunas alinhadas à direita e as temperaturas em Celsius devem ser precedidas por um sinal tanto para valores positivos como negativos.
- 15.13** Em algumas linguagens de programação, as strings são inseridas entre aspas simples ou duplas. Escreva um programa que lê as três strings `suzy`, `"suzy"` e `'suzy'`. As aspas simples e duplas são ignoradas ou lidas como parte da string?
- 15.14** Na Figura 11.5, os operadores de extração e inserção de fluxo foram sobrecarregados para entrada e saída de objetos da classe `PhoneNumber`. Reescreva o operador de extração de fluxo para realizar a seguinte verificação de erros na entrada. A função `operator>>` precisará ser reimplementada.
- Insira o número de telefone inteiro em um array. Teste se o número adequado de caracteres foi inserido. Deve haver um total de 14 caracteres lidos para um número de telefone na forma (800) 555-1212. Utilize a função-membro `ios_base::clear` para configurar `failbit` para a entrada imprópria.
  - O código de área e o prefixo não iniciam com 0 ou 1. Teste o primeiro dígito da parte do código da área e da parte do prefixo do número de telefone para certificar-se de que nenhum deles inicia com 0 ou 1. Utilize a função-membro `ios_base::clear` para configurar `failbit` para a entrada imprópria.
  - O dígito do meio de um código de área costumava ser limitado a 0 ou 1 (embora isso tenha mudado recentemente). Teste o dígito do meio quanto a um valor de 0 ou 1. Utilize a função-membro `ios_base::clear` para configurar `failbit` para a entrada imprópria. Se nenhuma das operações acima resultar na configuração de `failbit` para uma entrada imprópria, copie as três partes do número de telefone para os membros `areaCode` (código de área), `exchange` (prefixo da região) e `line` (linha) do objeto `PhoneNumber`. No programa principal, se `failbit` foi configurado na entrada, faça o programa imprimir uma mensagem de erro e terminar, em vez de imprimir o número de telefone.
- 15.15** Escreva um programa que realiza cada uma das seguintes tarefas:
- Crie uma classe definida pelo usuário `Point` que contenha os membros de dados `private` do tipo inteiro `xCoordinate` e `yCoordinate` e declare as funções operadoras sobrecarregadas de inserção e extração de fluxo como `friends` da classe.
  - Defina as funções operadoras de inserção e extração de fluxo. A função operadora de extração de fluxo deve determinar se os dados inseridos são válidos e, se não forem, deve configurar o `failbit` para indicar entrada inválida. O operador de inserção de fluxo não deve conseguir exibir o ponto depois que um erro de entrada tiver ocorrido.
  - Escreva uma função `main` que testa a entrada e a saída da classe definida pelo usuário `Point`, utilizando operadores sobrecarregados de extração e inserção de fluxo.
- 15.16** Escreva um programa que realiza cada uma das seguintes tarefas:
- Crie uma classe definida pelo usuário `Complex` que contenha os membros de dados `private` do tipo inteiro `real` e `imaginary` e declare as funções operadoras sobrecarregadas de inserção e extração de fluxo como `friends` da classe.
  - Defina as funções operadoras de inserção e extração de fluxo. A função operadora de extração de fluxo deve determinar se os dados inseridos são válidos e, se não o forem, deve configurar `failbit` para indicar uma entrada inválida. A entrada deve ser na forma
- $$3 + 8i$$
- Os valores podem ser negativos ou positivos e é possível que um dos dois valores não seja fornecido. Se um valor não for fornecido, o membro de dados apropriado deve ser configurado como 0. O operador de inserção de fluxo não deve ser capaz de exibir o ponto se um erro de entrada ocorrer. Para os valores imaginários negativos, deve-se imprimir um sinal de subtração em vez de um sinal de adição.
  - Escreva uma função `main` que testa a entrada e a saída da classe definida pelo usuário `Complex`, utilizando operadores sobrecarregados de extração e inserção de fluxo.
- 15.17** Escreva um programa que utiliza uma instrução `for` para imprimir uma tabela de valores ASCII para os caracteres do conjunto de caracteres ASCII de 33 a 126. O programa deve imprimir os valores decimal, octal e hexadecimal e o valor de caractere para cada caractere. Utilize os manipuladores de fluxo `dec`, `oct` e `hex` para imprimir os valores do tipo inteiro.
- 15.18** Escreva um programa para mostrar que as funções-membro `getline` e `getistream` de três argumentos terminam a string de entrada com um caractere de terminação de string nulo. Além disso, mostre que `get` deixa o caractere delimitador no fluxo de entrada, enquanto `getline` extrai o caractere delimitador e o descarta. O que acontece com os caracteres não lidos no fluxo?



*Nunca me esqueço de um rosto,  
mas no seu caso farei uma  
exceção.*

Groucho Marx

*É do senso comum capturar  
um método e experimentá-lo.  
Se ele falhar, admita isso com  
franqueza e experimente outro.  
Mas acima de tudo, tente algo.*

Franklin Delano Roosevelt

*Jogai fora a metade que não  
presta, para com a outra parte  
serdes pura.*

William Shakespeare

*Se eles estiverem correndo e não  
olharem para onde estão indo  
Tenho de sair de algum lugar e  
capturá-los.*

Jerome David Salinger

*Rei dos reis, heroísmo sem  
limites, sorridente escapaste da  
cilada gigantesca do mundo?*

William Shakespeare

## Tratamento de exceções

### OBJETIVOS

Neste capítulo, você aprenderá:

- O que são exceções e quando utilizá-las.
- A utilizar try, catch e throw para detectar, indicar e tratar exceções, respectivamente.
- A processar exceções não interceptadas e inesperadas.
- Como declarar novas classes de exceção.
- Como o desempilhamento permite que exceções não capturadas em um escopo sejam capturadas em outro escopo.
- A tratar falhas new.
- Como utilizar auto\_ptr para evitar vazamentos de memória.
- A entender a hierarquia de exceções-padrão.



- 16.1** Introdução
- 16.2** Visão geral do tratamento de exceções
- 16.3** Exemplo: tratando uma tentativa de divisão por zero
- 16.4** Quando utilizar o tratamento de exceções
- 16.5** Relançando uma exceção
- 16.6** Especificações de exceção
- 16.7** Processando exceções inesperadas
- 16.8** Desempilhamento de pilha
- 16.9** Construtores, destrutores e tratamento de exceções
- 16.10** Exceções e herança
- 16.11** Processando falhas new
- 16.12** Classe `auto_ptr` e alocação de memória dinâmica
- 16.13** Hierarquia de exceções da biblioteca-padrão
- 16.14** Outras técnicas de tratamento de erro
- 16.15** Síntese

Resumo | Terminologia | Exercícios de revisão | Respostas dos exercícios de revisão | Exercícios

## 16.1 Introdução

Neste capítulo, apresentamos o **tratamento de exceções**. Uma **exceção** é uma indicação de um problema que ocorre durante a execução de um programa. O nome ‘exceção’ dá a entender que o problema ocorre raramente — se a ‘regra’ é que uma instrução execute geralmente de modo correto, então a ‘exceção à regra’ é que um problema ocorra. O tratamento de exceções permite aos programadores criar aplicativos que podem resolver (ou tratar) exceções. Em muitos casos, o tratamento de uma exceção permite que um programa continue executando como se nenhum problema tivesse sido encontrado. Um problema mais grave poderia impedir um programa de continuar executando normalmente, em vez de requerer que o programa notificasse o usuário sobre o problema antes de terminar de uma maneira controlada. Os recursos apresentados neste capítulo permitem aos programadores escrever **programas robustos e tolerantes a falhas** capazes de lidar com possíveis problemas e continuar executando ou terminar elegantemente. O estilo e os detalhes de tratamento de exceções do C++ são baseados, em parte, no trabalho de Andrew Koenig e Bjarne Stroustrup, tal como apresentado em seu artigo, “Exception Handling for C++ (revisado)”.<sup>1</sup>



### Dica de prevenção de erro 16.1

*O tratamento de exceções ajuda a aprimorar a tolerância a falhas de um programa.*



### Observação de engenharia de software 16.1

*O tratamento de exceções fornece um mecanismo-padrão para processar erros. Isso é especialmente importante ao trabalhar em um projeto com uma grande equipe de programadores.*

O capítulo começa com uma visão geral de conceitos de tratamento de exceções e, em seguida, demonstra técnicas básicas de tratamento de exceções. Apresentamos essas técnicas via um exemplo que demonstra o tratamento de uma exceção que ocorre quando uma função tenta fazer uma divisão por zero. Depois discutimos questões adicionais sobre o tratamento de exceções como, por exemplo, a maneira de tratar exceções que ocorrem em um construtor ou destrutor e a maneira de tratar exceções que ocorrem se o operador `new` não consegue alocar memória para um objeto. Concluímos o capítulo introduzindo várias classes fornecidas pela C++ Standard Library para tratar exceções.

## 16.2 Visão geral do tratamento de exceções

A lógica do programa frequentemente testa condições que determinam como a execução de programa prossegue. Considere o seguinte pseudocódigo:

<sup>1</sup> Koenig, A., e B. Stroustrup. “Exception Handling for C++ (revised)”, *Proceedings of the Usenix C++ Conference*, p. 149–176, San Francisco, abril de 1990.



*Realize uma tarefa*

*Se a tarefa anterior não tiver sido executada corretamente*

*Realize processamento de erro*

*Realize a próxima tarefa*

*Se a tarefa anterior não tiver sido executada corretamente*

*Realize processamento de erro*

...

Nesse pseudocódigo, começamos executando uma tarefa. Então testamos se essa tarefa foi executada corretamente. Se não tiver sido, realizamos processamento de erro. Caso contrário, continuamos com a próxima tarefa. Embora essa forma de tratamento de erro funcione, mesclar o programa com a lógica de tratamento de erro pode dificultar a leitura, modificação, manutenção e depuração do programa — especialmente em aplicativos grandes.



### Dica de desempenho 16.1

*Se os problemas potenciais ocorrem raramente, misturar a lógica do programa e a lógica do tratamento de erro pode degradar o desempenho de um programa, porque o programa deve realizar testes (potencialmente freqüentes) para determinar se a tarefa foi executada corretamente e se a próxima tarefa pode ser realizada.*

O tratamento de exceções permite que o programador remova da ‘linha principal’ de execução do programa o código de tratamento de erro, o que aprimora a clareza do programa e fortalece sua modificabilidade. Os programadores podem decidir tratar a exceção que eles escolherem — todas as exceções, todas as exceções de certo tipo ou todas as exceções de um grupo de tipos relacionados (por exemplo, tipos de exceção que pertencem a uma hierarquia de herança). Tal flexibilidade reduz a probabilidade de que os erros não sejam percebidos, tornando, assim, um programa mais robusto.

Com as linguagens de programação que não suportam tratamento de exceções, os programadores costumam demorar em escrever código de processamento de erro ou, às vezes, se esquecem de incluí-lo. Isso resulta em produtos de software menos robustos. O C++ permite ao programador lidar facilmente com o tratamento de exceções desde o começo de um projeto.

## 16.3 Exemplo: tratando uma tentativa de divisão por zero

Vamos considerar um exemplo de tratamento de exceções simples (figuras 16.1–16.2). O propósito desse exemplo é impedir um problema aritmético comum — a divisão por zero. No C++, a divisão por zero utilizando a aritmética de inteiro geralmente faz com que um programa termine prematuramente. Na aritmética de ponto flutuante, a divisão por zero é permitida — ela resulta em positivo ou negativo infinito, que é exibido como INF ou -INF.

Nesse exemplo, definimos uma função chamada `quotient` que recebe dois inteiros inseridos pelo usuário e divide seu primeiro parâmetro `int` por seu segundo parâmetro `int`. Antes de realizar a divisão, a função faz coerção do valor do primeiro parâmetro `int` para o tipo `double`. Então, o valor do segundo parâmetro `int` é promovido para o tipo `double` para realizar o cálculo. Portanto, a função `quotient` realmente realiza a divisão utilizando dois valores `double` e retorna um resultado `double`.

Embora a divisão por zero seja permitida na aritmética de ponto flutuante, para o propósito desse exemplo, tratamos qualquer tentativa de divisão por zero como um erro. Portanto, a função `quotient` testa seu segundo parâmetro para assegurar que ele não é zero antes de permitir a continuação da divisão. Se o segundo parâmetro for zero, a função utiliza uma exceção a fim de indicar para o chamador que ocorreu um problema. O chamador (`main` neste exemplo) pode então processar essa exceção e permite que o usuário digite dois novos valores antes de chamar a função `quotient` novamente. Dessa maneira, o programa pode continuar a executar mesmo depois de um valor impróprio ser inserido, tornando, assim, o programa mais robusto.

O exemplo consiste em dois arquivos — `DivideByZeroException.h` (Figura 16.1) define uma classe de exceções que representa o tipo do problema que talvez ocorra no exemplo, e `fig16_02.cpp` (Figura 16.2) define a função `quotient` e a função `main` que a chama. A função `main` contém o código que demonstra o tratamento de exceções.

### Definindo uma classe de exceções para representar o tipo de problema que poderia ocorrer

A Figura 16.1 define a classe `DivideByZeroException` como uma classe derivada de classe da Standard Library `runtime_error` (definida no arquivo de cabeçalho `<stdexcept>`). A classe `runtime_error` — uma classe derivada da classe `exception` da Standard Library (definida no arquivo de cabeçalho `<exception>`) — é a classe básica padrão C++ para representar erros de tempo de execução. A classe `exception` é a classe básica padrão C++ para todas as exceções. (A Seção 16.13 discute a classe `exception` e suas classes derivadas detalhadamente.) Uma classe de exceções típica que deriva da classe `runtime_error` define somente um construtor (por exemplo, linhas 12–13) que passa uma string de mensagem de erro ao construtor `runtime_error` da classe básica. Cada classe de exceções que deriva direta ou indiretamente de `exception` contém a função `virtual what`, que retorna uma mensagem de erro do objeto de exceção. Observe que você não é solicitado a derivar uma classe de exceções personalizada, como `DivideByZeroException`, das classes de exceções-padrão fornecidas pelo C++. Entretanto, fazer isso permite aos programadores utilizar a função `virtual what` para obter uma mensagem de erro apropriada. Utilizamos um objeto dessa classe `DivideByZeroException` na Figura 16.2 para indicar quando uma tentativa de divisão por zero é feita.

```

1 // Figura 16.1: DivideByZeroException.h
2 // Definição da classe DivideByZeroException.
3 #include <stdexcept> // arquivo de cabeçalho stdexcept contém runtime_error
4 using std::runtime_error; // classe runtime_error da biblioteca-padrão do C++
5
6 // objetos DivideByZeroException devem ser lançados por funções
7 // ao detectar exceções de divisão por zero
8 class DivideByZeroException : public runtime_error
9 {
10 public:
11 // construtor especifica a mensagem de erro-padrão
12 DivideByZeroException::DivideByZeroException()
13 : runtime_error("attempted to divide by zero") {}
14 }; // fim da classe DivideByZeroException

```

**Figura 16.1** Definição da classe DivideByZeroException.

### *Demonstrando o tratamento de exceções*

O programa da Figura 16.2 usa o tratamento de exceções para empacotar o código que poderia lançar uma exceção de ‘divisão por zero’ e para tratar essa exceção, se ela ocorrer. O aplicativo permite ao usuário inserir dois inteiros, que são passados como argumentos para a função `quotient` (linhas 13–21). Essa função divide o primeiro número (`numerator`) pelo segundo (`denominator`). Supondo que o usuário não especificasse 0 como o denominador para a divisão, a função `quotient` retornaria o resultado de divisão. Entretanto, se o usuário inserir um valor 0 como o denominador, a função `quotient` lançará uma exceção. Na saída de exemplo, as duas primeiras linhas mostram um cálculo bem-sucedido e as duas próximas linhas mostram um cálculo que falhou devido a uma tentativa de dividir por zero. Quando a exceção ocorre, o programa informa o erro ao usuário e solicita que o usuário insira dois novos inteiros. Depois de discutirmos o código, consideraremos as entradas de usuário e o fluxo de controle de programa que resulta nessas saídas.

### *Incluindo código em um bloco **try***

O programa começa pedindo ao usuário para inserir dois inteiros. Os inteiros são inseridos na condição do loop `while` (linha 32). Depois que o usuário insere os valores que representam o numerador e o denominador, o controle de programa prossegue no corpo do loop (linhas 33–50). A linha 38 passa esses valores para a função `quotient` (linhas 13–21), que divide os inteiros e retorna um resultado ou **lança uma exceção** (isto é, indica que ocorreu um erro) em uma tentativa de dividir por zero. O tratamento de exceções é projetado para situações em que a função que detecta um erro é incapaz de tratá-lo.

O C++ fornece **blocos try** para permitir o tratamento de exceções. Um bloco `try` consiste na palavra-chave **try** seguida por chaves `{}` que definem um bloco de código em que as exceções poderiam ocorrer. O bloco `try` inclui instruções que poderiam causar exceções e instruções que devem ser ignoradas se ocorrer uma exceção.

Observe que um bloco `try` (linhas 36–40) inclui a invocação de função `quotient` e a instrução que exibe o resultado da divisão. Neste exemplo, como a invocação para a função `quotient` (linha 38) pode lançar uma exceção, incluímos essa invocação de função dentro de um bloco `try`. Incluir a instrução de saída (linha 39) no bloco `try` assegura que a saída só ocorrerá se a função `quotient` retornar um resultado.



### **Observação de engenharia de software 16.2**

*As exceções podem emergir pelo código explicitamente mencionado em um bloco `try`, por meio de chamadas a outras funções e por meio de chamadas de função profundamente aninhadas iniciadas pelo código em um bloco `try`.*

```

1 // Figura 16.2: Fig16_02.cpp
2 // Um exemplo simples de tratamento de exceções que verifica
3 // exceções de divisão por zero.
4 #include <iostream>
5 using std::cin;
6 using std::cout;
7 using std::endl;

```

**Figura 16.2** O exemplo de tratamento de exceções que lança exceções nas tentativas de divisão por zero.

(continua)

```

8
9 #include "DivideByZeroException.h" // classe DivideByZeroException
10
11 // realiza a divisão e lança o objeto DivideByZeroException se
12 // a exceção de divisão por zero ocorrer
13 double quotient(int numerator, int denominator)
14 {
15 // lança DivideByZeroException se tentar dividir por zero
16 if (denominator == 0)
17 throw DivideByZeroException(); // termina a função
18
19 // retorna resultado da divisão
20 return static_cast< double >(numerator) / denominator;
21 } // fim da função quotient
22
23 int main()
24 {
25 int number1; // numerador especificado pelo usuário
26 int number2; // denominador especificado pelo usuário
27 double result; // resultado da divisão
28
29 cout << "Enter two integers (end-of-file to end): ";
30
31 // permite ao usuário inserir dois inteiros para dividir
32 while (cin >> number1 >> number2)
33 {
34 // bloco try contém código que poderia lançar exceção
35 // e código que não deve executar se uma exceção ocorrer
36 try
37 {
38 result = quotient(number1, number2);
39 cout << "The quotient is: " << result << endl;
40 } // fim do try
41
42 // handler de exceção trata uma exceção de divisão por zero
43 catch (DivideByZeroException ÷ByZeroException)
44 {
45 cout << "Exception occurred: "
46 << divideByZeroException.what() << endl;
47 } // fim do catch
48
49 cout << "\nEnter two integers (end-of-file to end): ";
50 } // fim do while
51
52 cout << endl;
53 return 0; // termina normalmente
54 } // fim de main

```

```

Enter two integers (end-of-file to end): 100 7
The quotient is: 14.2857

```

```

Enter two integers (end-of-file to end): 100 0
Exception occurred: attempted to divide by zero

```

```

Enter two integers (end-of-file to end): ^Z

```

**Figura 16.2** O exemplo de tratamento de exceções que lança exceções nas tentativas de divisão por zero.

(continuação)

### Definindo um handler **catch** para processar uma **DivideByZeroException**

As exceções são processadas por **handlers catch** (também chamados de **handlers de exceção**), que capturam e tratam exceções. Pelo menos um handler catch (linhas 43–47) deve imediatamente seguir cada bloco try. Cada handler catch inicia com a palavra-chave **catch** e especifica em parênteses um **parâmetro de exceção** que representa o tipo de exceção que o handler catch pode processar (**DivideByZeroException** neste caso). Quando ocorre uma exceção em um bloco try, o handler catch que executa é aquele cujo tipo corresponde ao tipo da exceção que ocorreu (isto é, o tipo no bloco catch corresponde exatamente ao tipo de exceção lançado ou é uma classe básica dele). Se um parâmetro de exceção inclui um nome de parâmetro opcional, o handler catch pode utilizar esse nome para interagir com um objeto de exceção capturado no corpo do handler catch, que é delimitado por chaves (**{** e **}**). Em geral, um handler catch informa ao usuário sobre o erro, registra esse erro em um arquivo de log, termina o programa elegantemente ou tenta uma estratégia alternativa para realizar a tarefa que falhou. Neste exemplo, o handler catch simplesmente informa que o usuário tentou dividir por zero. Então o programa solicita que o usuário insira dois novos valores de inteiro.



#### Erro comum de programação 16.1

*É um erro de sintaxe colocar código entre um bloco try e seus blocos catch correspondentes.*



#### Erro comum de programação 16.2

*Cada handler catch pode ter apenas um único parâmetro — especificar uma lista de parâmetros de exceção separados por vírgulas é um erro de sintaxe.*



#### Erro comum de programação 16.3

*É um erro de lógica capturar o mesmo tipo em dois handlers catch diferentes que se seguem a um único bloco try.*

### Modelo de terminação do tratamento de exceções

Se ocorrer uma exceção como o resultado de uma instrução em um bloco try, o bloco try expira (isto é, termina imediatamente). Em seguida, o programa procura o primeiro handler catch que pode processar o tipo de exceção que ocorreu. O programa localiza o catch correspondente comparando o tipo de exceção lançado com o tipo de parâmetro de exceção de cada catch até encontrar uma correspondência. Ocorre uma correspondência se os tipos forem idênticos ou se o tipo da exceção lançada for uma classe derivada do tipo de parâmetro de exceção. Quando ocorre uma correspondência, o código contido no handler catch correspondente executa. Quando um handler catch termina o processamento alcançando sua chave de fechamento direita (**}**), a exceção é considerada tratada, e as variáveis locais definidas dentro do handler catch (incluindo o parâmetro catch) saem do escopo. O controle de programa não retorna ao ponto em que a exceção ocorreu (conhecido como o **ponto de lançamento**), porque o bloco try expirou. Em vez disso, o controle retoma a primeira instrução (linha 49) depois do último handler catch que se segue ao bloco try. Isso é conhecido como o **modelo de terminação do tratamento de exceções**. [Nota: Algumas linguagens utilizam o **modelo de retomada do tratamento de exceções**, em que, depois de uma exceção ser tratada, o controle é retomado logo após o ponto de lançamento.] Como com qualquer outro bloco de código, quando um bloco try termina, as variáveis locais definidas no bloco saem de escopo.



#### Erro comum de programação 16.4

*Erros de lógica podem ocorrer se você assumir que, depois de uma exceção ser tratada, o controle retornará à primeira instrução depois do ponto de lançamento.*



#### Dica de prevenção de erro 16.2

*Com o tratamento de exceções, um programa pode continuar executando (em vez de encerrar) depois de lidar com um problema. Isso ajuda a assegurar o tipo de aplicativos robustos que colaboram para o que é chamado de computação de missão crítica ou computação de negócios críticos.*

Se o bloco try completar sua execução com sucesso (isto é, não ocorrerem exceções no bloco try), então o programa ignora os handlers catch e o controle do programa continua com a primeira instrução depois do último catch que se segue a esse bloco try. Se não ocorrer nenhuma exceção em um bloco try, o programa ignora o(s) handler(s) catch para esse bloco.

Se uma exceção que ocorre em um bloco try não tiver nenhum handler catch correspondente, ou se ocorrer uma exceção em uma instrução que não está em um bloco try, a função que contém a instrução termina imediatamente e o programa tenta localizar um bloco try envolvente na função chamadora. Esse processo é chamado **desempilhamento** e é discutido na Seção 16.8.

*Fluxo de controle de programa quando o usuário insere um denominador não-zero*

Considere o fluxo de controle quando o usuário insere o numerador 100 e o denominador 7 (isto é, as duas primeiras linhas de saída na Figura 16.2). Na linha 16, a função `quotient` determina que o denominador não é igual a zero, então a linha 20 realiza a divisão e retorna o resultado (14.2857) à linha 38 como um `double` (o `static_cast< double >` na linha 20 assegura o tipo de valor de retorno adequado). O controle de programa então continua sequencialmente a partir da linha 38, portanto, a linha 39 exibe o resultado da divisão e a linha 40 é o fim do bloco `try`. Como o bloco `try` foi concluído com sucesso e não lançou uma exceção, o programa não executa as instruções contidas no handler `catch` (linhas 43–47) e o controle continua com a linha 49 (a primeira linha de código depois do handler `catch`), que solicita ao usuário inserir mais dois inteiros.

*Fluxo de controle de programa quando o usuário insere um denominador zero*

Agora vamos considerar um caso mais interessante em que o usuário insere o numerador 100 e o denominador 0 (isto é, a terceira e quarta linhas de saída na Figura 16.2). Na linha 16, `quotient` determina que o denominador é igual a zero, que indica uma tentativa de dividir por zero. A linha 17 lança uma exceção, que representamos como um objeto da classe `DivideByZeroException` (Figura 16.1).

Observe que, para lançar uma exceção, a linha 17 utiliza a palavra-chave **throw** seguida por um operando que representa o tipo de exceção a lançar. Normalmente, uma instrução `throw` especifica um operando. (Na Seção 16.5, discutimos como utilizar uma instrução `throw` que não especifica operandos.) O operando de um `throw` pode ser de qualquer tipo. Se o operando for um objeto, chamamos esse operando de **objeto de exceção** — neste exemplo, o objeto de exceção é um objeto de tipo `DivideByZeroException`. Entretanto, um operando `throw` também pode assumir outros valores, como o valor de uma expressão (por exemplo, `throw x > 5`) ou o valor de um `int` (por exemplo, `throw 5`). Os exemplos neste capítulo focalizam exclusivamente o lançamento de objetos de exceção.

**Erro comum de programação 16.5**

*Seja atencioso ao lançar (throwing) o resultado de uma expressão condicional (`?:`), porque as regras de promoção poderiam fazer com que o valor fosse de um tipo diferente do esperado. Por exemplo, ao lançar um `int` ou um `double` da mesma expressão condicional, a expressão condicional converte o `int` em um `double`. Entretanto, o handler `catch` sempre captura o resultado como um `double`, em vez de capturá-lo como um `double` quando um `double` é lançado, e capturá-lo como um `int` quando um `int` é lançado.*

Como parte do lançamento de uma exceção, o operando `throw` é criado e utilizado para inicializar o parâmetro no handler `catch`, que discutiremos em breve. Neste exemplo, a instrução `throw` na linha 17 cria um objeto da classe `DivideByZeroException`. Quando a linha 17 lança a exceção, a função `quotient` encerra imediatamente. Portanto, a linha 17 lança a exceção *antes* de a função `quotient` poder realizar a divisão na linha 20. Essa é uma característica central do tratamento de exceções: uma função deve lançar uma exceção antes que o erro tenha uma oportunidade de ocorrer.

Como decidimos incluir a invocação da função `quotient` (linha 38) em um bloco `try`, o controle do programa entra no handler `catch` (linhas 43–47) que imediatamente se segue ao bloco `try`. Esse handler `catch` serve como o handler de exceção para a exceção de divisão por zero. Em geral, quando uma exceção é lançada dentro de um bloco `try`, é capturada por um handler `catch` que especifica o tipo que corresponde à exceção lançada. Nesse programa, o handler `catch` especifica que ele captura objetos `DivideByZeroException` — esse tipo corresponde ao tipo de objeto lançado na função `quotient`. Na realidade, o handler `catch` captura uma referência ao objeto `DivideByZeroException` criado pela instrução `throw` da função `quotient` (linha 17).

**Dica de desempenho 16.2**

*Capturar um objeto de exceção por referência elimina o overhead de copiar o objeto que representa a exceção lançada.*

**Boa prática de programação 16.1**

*Associar cada tipo de erro de tempo de execução com um objeto de exceção adequadamente nomeado aumenta a clareza do programa.*

O corpo do handler `catch` (linhas 45–46) imprime a mensagem de erro associada que é retornada chamando a função `what` da classe básica `runtime_error`. Essa função retorna a string que o construtor `DivideByZeroException` (linhas 12–13 na Figura 16.1) passou para o construtor da classe básica `runtime_error`.

**16.4 Quando utilizar o tratamento de exceções**

O tratamento de exceções é um processo projetado para **erros síncronos**, que ocorrem quando uma instrução executa. Exemplos comuns desses erros são subscritos de array fora do intervalo, overflow aritmético (isto é, um valor fora do intervalo representável de valores), divisão por zero, parâmetros de função inválidos e alocação de memória malsucedida (por causa da falta de memória). O tratamento de exceções não é projetado para processar erros associados com os eventos **assíncronos** (por exemplo, completamentos de E/S de disco, chegadas de mensagem de rede, cliques de mouse e pressionamentos de tecla), que ocorrem em paralelo com o, e independente do, fluxo de controle do programa.





### Observação de engenharia de software 16.3

*Incorpore sua estratégia de tratamento de exceções no sistema desde o princípio do processo de projeto. Pode ser difícil incluir um tratamento de exceções eficiente depois que um sistema foi implementado.*



### Observação de engenharia de software 16.4

*O tratamento de exceções fornece uma técnica única e uniforme para processamento de problemas. Isso ajuda os programadores em grandes projetos a entender o código de processamento de erro uns dos outros.*



### Observação de engenharia de software 16.5

*Evite utilizar o tratamento de exceções como uma forma alternativa de fluxo de controle. Essas exceções 'adicionais' podem 'entrar no caminho' de verdadeiras exceções do tipo erro.*



### Observação de engenharia de software 16.6

*O tratamento de exceções simplifica a combinação de componentes de software e permite trabalhar em conjunto eficientemente possibilitando que os componentes predefinidos comuniquem problemas para componentes específicos ao aplicativo, que então podem processar os problemas de maneira específica ao aplicativo.*

O mecanismo de tratamento de exceções também é útil para processar problemas que ocorrem quando um programa interage com elementos de software, como funções-membro, construtores, destrutores e classes. Em vez de tratar problemas internamente, tais elementos de software muitas vezes utilizam exceções para notificar aos programas quando problemas ocorrem. Isso permite aos programadores implementar tratamento personalizado de erros para cada aplicativo.



### Dica de desempenho 16.3

*Quando não ocorrer nenhuma exceção, o código de tratamento de exceções incorre em nenhuma ou em poucas penalidades de desempenho. Portanto, os programas que implementam o tratamento de exceções operam com mais eficiência do que os programas que mesclam o código de tratamento de erros com a lógica do programa.*



### Observação de engenharia de software 16.7

*As funções com condições de erro comuns devem retornar 0 ou NULL (ou outros valores adequados) em vez de lançar exceções. Um programa que chama tal função pode verificar o valor de retorno para determinar o sucesso ou a falha da chamada de função.*

Os aplicativos complexos normalmente consistem em componentes de software predefinidos e componentes específicos ao aplicativo que utilizam os componentes predefinidos. Ao encontrar um problema, o componente predefinido precisa de um mecanismo para comunicar o problema ao componente específico de aplicativo — o componente predefinido não prevê de que maneira cada aplicativo processa um problema que ocorre.

## 16.5 Relançando uma exceção

É possível que um handler de exceção, no recebimento de uma exceção, decida que não pode processar essa exceção ou que pode processá-la apenas parcialmente. Nesses casos, o handler de exceção pode adiar o tratamento de exceções (ou talvez uma parte dele) para outro handler de exceção. Em qualquer caso, o handler alcança isso **relançando a exceção** via a instrução

**throw;**

Independentemente de um handler poder ou não processar (mesmo parcialmente) uma exceção, o handler pode relançá-la para processamento adicional fora do handler. O próximo bloco try envolvente detecta a exceção relançada, que um handler catch listado depois desse bloco try envolvente tenta tratar.



### Erro comum de programação 16.6

*Executar uma instrução throw vazia situada fora de um handler catch produz uma chamada à função `terminate`, que abandona o processamento de exceção e termina o programa imediatamente.*

O programa da Figura 16.3 demonstra como relançar uma exceção. No bloco try de main (linhas 32–37), a linha 35 chama a função `throwException` (linhas 11–27). A função `throwException` também contém um bloco try (linhas 14–18), a partir do qual a instrução `throw` na linha 17 lança uma instância da classe `exception` da biblioteca-padrão. O handler catch da função `throwException` (linhas 19–24) captura essa exceção, imprime uma mensagem de erro (linhas 21–22) e relança a exceção (linha 23).

```

1 // Figura 16.3: Fig16_03.cpp
2 // Demonstrando o relançamento de exceção.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <exception>
8 using std::exception;
9
10 // lança, captura e relança a exceção
11 void throwException()
12 {
13 // lança a exceção e a captura imediatamente
14 try
15 {
16 cout << " Function throwException throws an exception\n";
17 throw exception(); // gera a exceção
18 } // fim do try
19 catch (exception &) // trata a exceção
20 {
21 cout << " Exception handled in function throwException"
22 << "\n Function throwException rethrows exception";
23 throw; // relança a exceção para processamento adicional
24 } // fim do catch
25
26 cout << "This also should not print\n";
27 } // fim da função throwException
28
29 int main()
30 {
31 // lança a exceção
32 try
33 {
34 cout << "\nmain invokes function throwException\n";
35 throwException();
36 cout << "This should not print\n";
37 } // fim do try
38 catch (exception &) // trata a exceção
39 {
40 cout << "\n\nException handled in main\n";
41 } // fim do catch
42
43 cout << "Program control continues after catch in main\n";
44 return 0;
45 } // fim de main

```

```

main invokes function throwException
Function throwException throws an exception
Exception handled in function throwException
Function throwException rethrows exception

```

```

Exception handled in main
Program control continues after catch in main

```

**Figura 16.3** Relançando uma exceção.



Após a instrução `throw` (linha 23), a função `throwException` é encerrada e retorna o controle para a linha 35 no bloco `try...catch` em `main`. O bloco `try` termina (portanto, a linha 36 não executa) e o handler `catch` em `main` (linhas 38–41) captura essa exceção e imprime uma mensagem de erro (linha 40). [Nota: Visto que não utilizamos os parâmetros de exceção nos handlers `catch` desse exemplo, omitimos os nomes de parâmetro de exceção e especificamos somente o tipo de exceção a capturar (linhas 19 e 38).]

## 16.6 Especificações de exceção

Uma **especificação de exceção** opcional (também chamada **lista `throw`**) enumera uma lista de exceções que uma função pode lançar. Por exemplo, considere a declaração de função

```
int someFunction(double value)
 throw (ExceptionA, ExceptionB, ExceptionC)
{
 // corpo da função
}
```

Nessa definição, a especificação de exceção, que inicia com a palavra-chave `throw` que se segue imediatamente ao parêntese de fechamento da lista de parâmetros de uma função, indica que a função `someFunction` pode lançar exceções de tipos `ExceptionA`, `ExceptionB` e `ExceptionC`. Uma função pode lançar somente exceções dos tipos indicados pela especificação ou exceções de qualquer tipo derivado desses tipos. Se a função lança uma exceção que não pertence a um tipo especificado, a função **unexpected** é chamada, o que normalmente termina o programa.

Uma função que não fornece uma especificação de exceção pode lançar qualquer exceção. Colocar `throw()` — uma **especificação de exceção vazia** — depois da lista de parâmetros de uma função declara que a função não lança exceções. Se a função tentar lançar uma exceção, a função `unexpected` é invocada. A Seção 16.7 mostra como a função `unexpected` pode ser personalizada chamando a função `set_unexpected`.



### Erro comum de programação 16.7

*Lançar uma exceção que não foi declarada em uma especificação de exceção da função produz uma chamada à função `unexpected`.*



### Dica de prevenção de erro 16.3

*O compilador não irá gerar um erro de compilação se uma função contiver uma expressão `throw` para uma exceção não listada na especificação de exceção da função. Um erro só ocorre quando essa função tenta lançar essa exceção em tempo de execução. Para evitar surpresas em tempo de execução, verifique cuidadosamente seu código a fim de assegurar que as funções não lancem exceções não listadas em suas especificações de exceção.*

## 16.7 Processando exceções inesperadas

A função `unexpected` chama a função registrada junto à função `set_unexpected` (definida no arquivo de cabeçalho `<exception>`). Se nenhuma função foi registrada dessa maneira, a função `terminate` é chamada por padrão. Os casos em que a função `terminate` é chamada incluem:

1. o mecanismo de exceção não pode localizar um `catch` correspondente para uma exceção lançada;
2. um destrutor tenta lançar (`throw`) uma exceção durante o desempilhamento;
3. uma tentativa de relançar uma exceção é feita quando não há nenhuma exceção sendo atualmente tratada;
4. uma chamada à função `unexpected` chama a função `terminate` por padrão.

(A Seção 15.5.1 do C++ Standard Document discute vários casos adicionais.) A função `set_terminate` pode especificar a função a ser invocada quando `terminate` for chamada. Caso contrário, `terminate` chama `abort`, que termina o programa sem chamar os destrutores de quaisquer objetos restantes da classe de armazenamento automático ou estático. Isso poderia levar a vazamentos de recurso quando um programa termina prematuramente.

A função `set_terminate` e a função `set_unexpected` retornam um ponteiro para a última função chamada por `terminate` e `unexpected`, respectivamente (0, na primeira vez que cada uma é chamada). Isso permite ao programador salvar o ponteiro de função para que possa ser restaurado mais tarde. As funções `set_terminate` e `set_unexpected` aceitam como argumentos ponteiros de funções com tipo de retorno `void` e sem argumentos.

Se a última ação de uma função de terminação definida pelo programador é não fechar um programa, a função `abort` será chamada para terminar a execução de programa depois que as outras instruções da função de terminação definida pelo programador forem executadas.

## 16.8 Desempilhamento de pilha

Quando uma exceção é lançada mas não capturada em um escopo particular, a pilha de chamadas de função é desempilhada e uma tentativa de capturar (catch) a exceção é feita no próximo bloco try externo. Desempilhar a pilha de chamadas de função significa que a função em que a exceção não foi capturada termina, todas as variáveis locais nessa função são destruídas e o controle retorna à instrução que originalmente invocou essa função. Se um bloco try incluir essa instrução, uma tentativa de capturar a exceção com catch é feita. Se um bloco try não incluir essa instrução, o desempilhamento ocorre novamente. Se nenhum handler catch capturar essa exceção, a função terminate é chamada para terminar o programa. O programa da Figura 16.4 demonstra o desempilhamento.

Em main, o bloco try (linhas 37–41) chama function1 (linhas 27–31). Em seguida, function1 chama function2 (linhas 20–24), que por sua vez chama function3 (linhas 11–17). A linha 16 de function3 lança um objeto runtime\_error. Entretanto, como nenhum bloco try inclui a instrução throw na linha 17, ocorre o desempilhamento — a function3 termina na linha 16 e, então, retorna o controle à instrução em function2 que invocou function3 (isto é, a linha 23). Como nenhum bloco try inclui a linha 23, o desempilhamento ocorre novamente — function2 termina na linha 24 e o controle retorna à instrução em function1 que invocou function2 (isto é, linha 30). Como nenhum bloco try inclui a linha 30, o desempilhamento ocorre mais uma vez — function1 termina na linha 31 e o controle retorna à instrução em main que invocou function1 (isto é, linha 40). O bloco try das linhas 37–41 inclui essa instrução, portanto o primeiro handler catch correspondente localizado depois desse bloco try (linha 42–46) captura e processa a exceção. A linha 44 utiliza a função what para exibir a mensagem de exceção. Lembre-se de que a função what é uma função virtual da classe exception que pode ser sobrescrita por uma classe derivada para retornar uma mensagem de erro adequada.

## 16.9 Construtores, destrutores e tratamento de exceções

Primeiro, vamos discutir uma questão que mencionamos, mas que ainda não resolvemos satisfatoriamente: O que acontece quando um erro é detectado em um construtor? Por exemplo, como o construtor de um objeto deve responder quando new falha porque foi incapaz de alocar a memória necessária para armazenar a representação interna desse objeto? Como o construtor não pode retornar um valor para indicar um erro, devemos escolher um meio alternativo de indicar que o objeto não foi construído adequadamente. Um esquema é retornar o objeto inadequadamente construído e esperar que qualquer pessoa que o utilize faça testes apropriados para determinar se ele está em um estado inconsistente. Outro esquema é configurar uma variável fora do construtor. Talvez a melhor alternativa seja requerer que o construtor lance uma exceção que contenha informações de erro, oferecendo, assim, uma oportunidade de o programa tratar a falha.

As exceções lançadas por um construtor fazem com que os destrutores sejam chamados por qualquer objeto construído como parte do objeto sendo construído antes de a exceção ser lançada. Os destrutores são chamados para cada objeto automático construído em um bloco try antes de uma exceção ser lançada. O desempilhamento garantidamente terá sido completado no ponto que um handler de exceção começar a executar. Se um destrutor invocado como resultado do desempilhamento lançar uma exceção, terminate é chamada.

Se um objeto tem objetos-membro e se uma exceção é lançada antes de o objeto externo ser completamente construído, então os destrutores serão executados para os objetos-membro que foram construídos antes da ocorrência da exceção. Se um array de objetos tiver sido parcialmente construído durante a ocorrência de uma exceção, somente os destrutores dos objetos construídos no array serão chamados.

Uma exceção poderia impedir a operação de código que normalmente liberaria um recurso, causando, assim, um vazamento de recurso. Uma técnica para resolver esse problema é inicializar um objeto local para adquirir o recurso. Quando uma exceção ocorre, o destrutor desse objeto será invocado e poderá liberar o recurso.



### Dica de prevenção de erro 16.4

*Quando uma exceção é lançada do construtor de um objeto que é criado em uma expressão new, a memória dinamicamente alocada desse objeto é liberada.*

## 16.10 Exceções e herança

Várias classes de exceções podem ser derivadas de uma classe básica comum, como discutimos na Seção 16.3, quando criamos a classe DivideByZeroException como uma classe derivada da classe exception. Se um handler catch captura um ponteiro ou referência para um objeto de exceção de um tipo de classe básica, ele também pode capturar um ponteiro ou referência para todos os objetos de classes publicamente derivadas dessa classe básica — isso permite processamento polimórfico de erros relacionados.



### Dica de prevenção de erro 16.5

*Utilizar herança com exceções permite um handler de exceção para capturar erros relacionados com a notação concisa. Uma abordagem é capturar cada tipo de ponteiro ou referência para um objeto de exceção de classe derivada individualmente, mas uma abordagem mais concisa é capturar as referências ou ponteiros para objetos de exceção de classe básica. Além disso, capturar ponteiros ou referências a objetos de exceção de classe derivada individualmente é propenso a erros, especialmente se o programador se esquecer de testar explicitamente um ou mais dos tipos de referência ou de ponteiros de classe derivada.*

```

1 // Figura 16.4: Fig16_04.cpp
2 // Demonstrando o desempilhamento.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <stdexcept>
8 using std::runtime_error;
9
10 // function3 lança erro de tempo de execução
11 void function3() throw (runtime_error)
12 {
13 cout << "In function 3" << endl;
14
15 // nenhum bloco try, o desempilhamento ocorre, retorna controle a function2
16 throw runtime_error("runtime_error in function3");
17 } // fim de function3
18
19 // function2 invoca function3
20 void function2() throw (runtime_error)
21 {
22 cout << "function3 is called inside function2" << endl;
23 function3(); // o desempilhamento ocorre, retorna o controle a function1
24 } // fim de function2
25
26 // function1 invoca function2
27 void function1() throw (runtime_error)
28 {
29 cout << "function2 is called inside function1" << endl;
30 function2(); // o desempilhamento ocorre, retorna controle a main
31 } // fim de function1
32
33 // demonstra o desempilhamento
34 int main()
35 {
36 // invoca function1
37 try
38 {
39 cout << "function1 is called inside main" << endl;
40 function1(); // chama function1 que lança runtime_error
41 } // fim do try
42 catch (runtime_error &error) // trata erro de tempo de execução
43 {
44 cout << "Exception occurred: " << error.what() << endl;
45 cout << "Exception handled in main" << endl;
46 } // fim do catch
47
48 return 0;
49 } // fim de main

```

```

function1 is called inside main
function2 is called inside function1
function3 is called inside function2
In function 3
Exception occurred: runtime_error in function3
Exception handled in main

```

**Figura 16.4** Desempilhamento.

## 16.11 Processando falhas new

O C++ padrão especifica que, quando o operador `new` falha, ele lança uma exceção `bad_alloc` (definida no arquivo de cabeçalho `<new>`). Entretanto, alguns compiladores não são compatíveis com o padrão C++ e, portanto, utilizam a versão de `new` que retorna 0 em caso de falha. Por exemplo, o Microsoft Visual Studio .NET lança uma exceção `bad_alloc` quando `new` falha, enquanto o Microsoft Visual C++ 6.0 retorna 0 em caso de falha de `new`.

Os compiladores variam em seu suporte para tratamento de falhas de `new`. Muitos compiladores C++ mais antigos retornam 0 por padrão quando `new` falha. Alguns compiladores suportam `new` lançando uma exceção se o arquivo de cabeçalho `<new>` (ou `<new.h>`) estiver incluído. Outros compiladores lançam `bad_alloc` por padrão, independentemente de o arquivo de cabeçalho `<new>` estar ou não incluído. Consulte a documentação do compilador para determinar o suporte do compilador para tratamento de falhas `new`.

Nesta seção, apresentamos três exemplos de falhas de `new`. O primeiro exemplo retorna 0 quando `new` falha. O segundo exemplo utiliza a versão de `new` que lança uma exceção `bad_alloc` quando `new` falha. O terceiro exemplo utiliza a função `set_new_handler` para tratar falhas de `new`. [Nota: Os exemplos nas figuras 16.5–16.7 alocam grandes quantidades de memória dinâmica, o que poderia fazer com que seu computador se tornasse lento.]

### *new* retornando 0 em caso de falha

A Figura 16.5 demonstra `new` retornando 0 em caso de falha em alocar a quantidade de memória solicitada. A instrução `for` nas linhas 13–24 devem fazer o loop 50 vezes e, em cada passagem, alocar um array de 50.000.000 valores `double` (isto é, 400.000.000 de bytes, porque um `double` tem normalmente 8 bytes). A instrução `if` na linha 17 testa o resultado de cada operação `new` para determinar se `new` alocou a memória com sucesso. Se `new` falhar e retornar 0, a linha 19 imprime uma mensagem de erro e o loop termina. [Nota: Utilizamos o Microsoft Visual C++ 6.0 para executar esse exemplo, porque ele lança uma exceção `bad_alloc` em caso de falha de `new` em vez de retornar 0.]

```

1 // Figura 16.5: Fig16_05.cpp
2 // Demonstrando new pré-padrão retornando 0 quando a memória
3 // não é alocada.
4 #include <iostream>
5 using std::cerr;
6 using std::cout;
7
8 int main()
9 {
10 double *ptr[50];
11
12 // aloca memória para ptr
13 for (int i = 0; i < 50; i++)
14 {
15 ptr[i] = new double[50000000];
16
17 if (ptr[i] == 0) // fez new falhar na alocação de memória
18 {
19 cerr << "Memory allocation failed for ptr[" << i << "]\n";
20 break;
21 } // fim do if
22 else // alocação bem-sucedida de memória
23 cout << "Allocated 50000000 doubles in ptr[" << i << "]\n";
24 } // fim do for
25
26 return 0;
27 } // fim de main

```

```

Allocated 50000000 doubles in ptr[0]
Allocated 50000000 doubles in ptr[1]
Allocated 50000000 doubles in ptr[2]
Memory allocation failed for ptr[3]

```

**Figura 16.5** new retornando 0 em caso de falha.

A saída mostra que o programa realizou somente três iterações antes de new falhar e o loop terminar. Sua saída poderia diferir com base na memória física, no espaço em disco disponível para memória virtual no sistema e no compilador em utilização.

### new lançando bad\_alloc em caso de falha

A Figura 16.6 demonstra new lançando bad\_alloc em caso de falha em alocar a memória solicitada. A instrução for (linhas 20–24) dentro do bloco try deve fazer um loop 50 vezes e, em cada passagem, alocar um array de 50.000.000 valores double. Se new falha e lança uma exceção bad\_alloc, o loop termina e o programa continua na linha 28, onde o handler catch captura e processa a exceção. As linhas 30–31 imprimem a mensagem "Exception occurred:" seguida pela mensagem retornada da versão da classe básica exception da função what (isto é, uma mensagem específica à exceção definida pela implementação, como "Allocation Failure" no Microsoft Visual Studio .NET 2003). A saída mostra que o programa realizou somente três iterações do loop antes de new falhar e lançar a exceção bad\_alloc. Sua saída poderia diferir com base na memória física, espaço em disco disponível para memória virtual no sistema e no compilador em utilização.

```

1 // Figura 16.6: Fig16_06.cpp
2 // Demonstrando new-padrão lançando bad_alloc quando a memória
3 // não pode ser alocada.
4 #include <iostream>
5 using std::cerr;
6 using std::cout;
7 using std::endl;
8
9 #include <new> // operador new padrão
10 using std::bad_alloc;
11
12 int main()
13 {
14 double *ptr[50];
15
16 // aloca memória para ptr
17 try
18 {
19 // aloca memória para ptr[i]; new lança bad_alloc em caso de falha
20 for (int i = 0; i < 50; i++)
21 {
22 ptr[i] = new double[50000000]; // pode lançar exceção
23 cout << "Allocated 50000000 doubles in ptr[" << i << "]\n";
24 } // fim do for
25 } // fim do try
26
27 // trata exceção bad_alloc
28 catch (bad_alloc &memoryAllocationException)
29 {
30 cerr << "Exception occurred: "
31 << memoryAllocationException.what() << endl;
32 } // fim do catch
33
34 return 0;
35 } // fim de main

```

```

Allocated 50000000 doubles in ptr[0]
Allocated 50000000 doubles in ptr[1]
Allocated 50000000 doubles in ptr[2]
Exception occurred: bad allocation

```

**Figura 16.6** new lançando bad\_alloc em caso de falha.

O C++ padrão especifica que compiladores compatíveis com o padrão podem continuar a utilizar uma versão de `new` que retorna 0 em caso de falha. Para esse propósito, o arquivo de cabeçalho `<new>` define o objeto `nothrow` (de tipo `nothrow_t`), que é utilizado como mostrado a seguir:

```
double *ptr = new(nothrow) double[50000000];
```

A instrução anterior utiliza a versão de `new` que não lança exceções `bad_alloc` (isto é, `nothrow`) para alocar um array de 50.000.000 doubles.



### Observação de engenharia de software 16.8

*Para tornar os programas mais robustos, utilize a versão de `new` que lança exceções `bad_alloc` em caso de falhas.*

#### Tratando falhas `new` com a função `set_new_handler`

Um recurso adicional para tratar falhas de `new` é a função `set_new_handler` (prototipada no arquivo de cabeçalho-padrão `<new>`). Essa função aceita como seu argumento um ponteiro para uma função que não aceita argumentos e retorna `void`. Esse ponteiro aponta para a função que será chamada se `new` falhar. Isso fornece ao programador uma abordagem uniforme para tratar todas as falhas de `new`, independentemente de onde uma falha ocorra no programa. Uma vez que `set_new_handler` registra um **handler de new** no programa, o operador `new` não lança `bad_alloc` em caso de falha; em vez disso, ele adia o tratamento de erro para a função do handler de `new`.

Se `new` aloca a memória com sucesso, ele retorna um ponteiro para essa memória. Se `new` não conseguir alocar memória e `set_new_handler` não tiver registrado uma função handler de `new`, `new` lançará uma exceção `bad_alloc`. Se `new` não conseguir alocar memória e uma função handler de `new` tiver sido registrada, a função handler de `new` é chamada. O C++ padrão especifica que a função handler de `new` deve realizar uma das seguintes tarefas:

1. Disponibilizar mais memória excluindo outra memória dinamicamente alocada (ou pedindo ao usuário para fechar outros aplicativos) e retornar ao operador `new` para tentar alocar memória novamente.
2. Lançar uma exceção de tipo `bad_alloc`.
3. Chamar a função `abort` ou `exit` (ambas localizadas no arquivo de cabeçalho `<cstdlib>`) para terminar o programa.

A Figura 16.7 demonstra `set_new_handler`. A função `customNewHandler` (linhas 14–18) imprime uma mensagem de erro (linha 16) e, então, termina o programa via chamada para `abort` (linha 17). A saída mostra que o programa realizou somente três iterações do loop antes de `new` falhar e invocar a função `customNewHandler`. Sua saída pode diferir com base na memória física, no espaço em disco disponível para memória virtual no sistema e no compilador utilizado para compilar o programa.

```
1 // Figura 16.7: Fig16_07.cpp
2 // Demonstrando set_new_handler.
3 #include <iostream>
4 using std::cerr;
5 using std::cout;
6
7 #include <new> // operador new-padrão e set_new_handler
8 using std::set_new_handler;
9
10 #include <cstdlib> // protótipo da função abort
11 using std::abort;
12
13 // trata falha de alocação de memória
14 void customNewHandler()
15 {
16 cerr << "customNewHandler was called";
17 abort();
18 } // fim da função customNewHandler
19
20 // utilizando set_new_handler para tratar alocação de memória malsucedida
21 int main()
22 {
23 double *ptr[50];
```

**Figura 16.7** `set_new_handler` especificando a chamada de função quando `new` falha.

(continua)

```

24
25 // especifica que customNewHandler deve ser chamado em
26 // caso de falha na alocação de memória
27 set_new_handler(customNewHandler);
28
29 // aloca memória para ptr[i]; customNewHandler será
30 // chamado na falha na alocação de memória
31 for (int i = 0; i < 50; i++)
32 {
33 ptr[i] = new double[50000000]; // pode lançar exceção
34 cout << "Allocated 50000000 doubles in ptr[" << i << "]\n";
35 } // fim do for
36
37 return 0;
38 } // fim de main

```

```

Allocated 50000000 doubles in ptr[0]
Allocated 50000000 doubles in ptr[1]
Allocated 50000000 doubles in ptr[2]
customNewHandler was called

```

**Figura 16.7** `set_new_handler` especificando a chamada de função quando `new` falha.

(continuação)

## 16.12 Classe `auto_ptr` e alocação de memória dinâmica

Uma prática de programação comum é alocar memória dinâmica, atribuir o endereço dessa memória a um ponteiro, utilizar o ponteiro para manipular a memória e desalocar a memória com `delete` quando ela não for mais necessária. Se ocorrer uma exceção depois da alocação de memória bem-sucedida, mas antes de a instrução `delete` executar, um vazamento de memória poderia ocorrer. O padrão C++ fornece o template da classe `auto_ptr` no arquivo de cabeçalho `<memory>` para lidar com essa situação.

Um objeto da classe `auto_ptr` mantém um ponteiro para a memória dinamicamente alocada. Quando um destrutor de objeto `auto_ptr` é chamado (por exemplo, quando um objeto `auto_ptr` sai de escopo), ele realiza uma operação `delete` em seu membro de dados de ponteiro. O template da classe `auto_ptr` fornece os operadores sobrecarregados `*` e `->` para que um objeto `auto_ptr` possa ser utilizado da mesma maneira que uma variável de ponteiro regular. A Figura 16.10 demonstra um objeto `auto_ptr` que aponta para um objeto dinamicamente alocado da classe `Integer` (figuras 16.8–16.9).

A linha 18 da Figura 16.10 cria objeto `auto_ptr ptrToInteger` e o inicializa com um ponteiro para um objeto `Integer` dinamicamente alocado que contém o valor 7. A linha 21 utiliza o operador `->` sobrecarregado `auto_ptr` para invocar a função `setInteger` no objeto `Integer` apontado por `ptrToInteger`. A linha 24 utiliza o operador `*` sobrecarregado `auto_ptr` para desreferenciar `ptrToInteger`, então utiliza o operador ponto (`.`) para invocar a função `getInteger` no objeto `Integer` apontado por `ptrToInteger`. Como um ponteiro regular, os operadores sobrecarregados `->` e `*` de um `auto_ptr` podem ser utilizados para acessar o objeto para o qual `auto_ptr` aponta.

Como `ptrToInteger` é uma variável automática local em `main`, `ptrToInteger` é destruído quando `main` termina. O destrutor `auto_ptr` força uma exclusão (`delete`) do objeto `Integer` apontado por `ptrToInteger`, que por sua vez chama o destrutor de classe `Integer`. A memória que `Integer` ocupa é liberada, independentemente de como o controle deixa o bloco (por exemplo, por uma instrução `return` ou por uma exceção). O mais importante é que o uso dessa técnica pode evitar vazamentos de memória. Por exemplo, suponha que uma função retorne um ponteiro apontado para algum objeto. Infelizmente, o chamador da função que recebe esse ponteiro poderia não excluir o objeto, resultando, assim, em um vazamento de memória. Entretanto, se a função retornar um `auto_ptr` ao objeto, o objeto será excluído automaticamente quando o destrutor do objeto `auto_ptr` for chamado.

Um `auto_ptr` pode passar a posse da memória dinâmica que ele gerencia via seu operador de atribuição sobrecarregado ou construtor de cópia. O último objeto `auto_ptr` que mantém o ponteiro para a memória dinâmica excluirá a memória. Isso torna `auto_ptr` um mecanismo ideal para retornar a memória alocada dinamicamente ao código-cliente. Quando o `auto_ptr` sai de escopo no código-cliente, o destrutor `auto_ptr` exclui a memória dinâmica.



### Observação de engenharia de software 16.9

Um `auto_ptr` tem restrições em certas operações. Por exemplo, um `auto_ptr` não pode apontar para um array ou uma classe contêiner padrão.



```

1 // Figura 16.8: Integer.h
2 // Definição da classe Integer.
3
4 class Integer
5 {
6 public:
7 Integer(int i = 0); // construtor-padrão Integer
8 ~Integer(); // destrutor Integer
9 void setInteger(int i); // função para configurar Integer
10 int getInteger() const; // função para retornar Integer
11 private:
12 int value;
13 }; // fim da classe Integer

```

**Figura 16.8** Definição da classe Integer.

```

1 // Figura 16.9: Integer.cpp
2 // Definição da função-membro Integer.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Integer.h"
8
9 // construtor-padrão Integer
10 Integer::Integer(int i)
11 : value(i)
12 {
13 cout << "Constructor for Integer " << value << endl;
14 } // fim do construtor Integer
15
16 // destrutor Integer
17 Integer::~Integer()
18 {
19 cout << "Destructor for Integer " << value << endl;
20 } // fim do destrutor Integer
21
22 // configura o valor Integer
23 void Integer::setInteger(int i)
24 {
25 value = i;
26 } // fim da função setInteger
27
28 // retorna o valor Integer
29 int Integer::getInteger() const
30 {
31 return value;
32 } // fim da função getInteger

```

**Figura 16.9** Definição de função-membro da classe Integer.

```

1 // Figura 16.10: Fig16_10.cpp
2 // Demonstrando auto_ptr.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <memory>
8 using std::auto_ptr; // definição da classe auto_ptr
9
10 #include "Integer.h"
11
12 // utiliza auto_ptr para manipular o objeto Integer
13 int main()
14 {
15 cout << "Creating an auto_ptr object that points to an Integer\n";
16
17 // "aponta" auto_ptr para o objeto Integer
18 auto_ptr< Integer > ptrToInteger(new Integer(7));
19
20 cout << "\nUsing the auto_ptr to manipulate the Integer\n";
21 ptrToInteger->setInteger(99); // usa auto_ptr para configurar o valor Integer
22
23 // utiliza auto_ptr para obter o valor Integer
24 cout << "Integer after setInteger: " << (*ptrToInteger).getInteger();
25 return 0;
26 } // fim de main

```

Creating an auto\_ptr object that points to an Integer  
 Constructor for Integer 7

Using the auto\_ptr to manipulate the Integer  
 Integer after setInteger: 99

Terminating program  
 Destructor for Integer 99

**Figura 16.10** O objeto auto\_ptr gerencia a memória dinamicamente alocada.

## 16.13 Hierarquia de exceções da biblioteca-padrão

A experiência mostrou que exceções entram satisfatoriamente em várias categorias. A C++ Standard Library inclui uma hierarquia de classes de exceções (Figura 16.11). Como discutido inicialmente na Seção 16.3, essa hierarquia é comandada pela classe básica `exception` (definida no arquivo de cabeçalho `<exception>`), que contém a função `virtual what`, que as classes derivadas podem sobrescrever para emitir mensagens de erro apropriadas.

As classes derivadas imediatas da classe básica `exception` incluem `runtime_error` e `logic_error` (ambas definidas no cabeçalho `<stdexcept>`), cada uma das quais possuindo várias classes derivadas. Também derivadas de `exception` são as exceções lançadas pelos operadores C++ — por exemplo, `bad_alloc` é lançada por `new` (Seção 16.11), `bad_cast` é lançada por `dynamic_cast` (Capítulo 13) e `bad_typeid` é lançada por `typeid` (Capítulo 13). Incluir `bad_exception` na lista `throw` de uma função significa que, se uma exceção inesperada ocorrer, a função `unexpected` pode lançar `bad_exception` em vez de terminar a execução do programa (por padrão) ou chamar outra função especificada por `set_unexpected`.



### Erro comum de programação 16.8

Colocar um handler `catch` que captura um objeto de classe básica antes de um `catch` que captura um objeto de uma classe derivada dessa classe básica é um erro de lógica. A classe básica `catch` captura todos os objetos de classes derivadas dessa classe básica, então a classe derivada `catch` nunca executará.

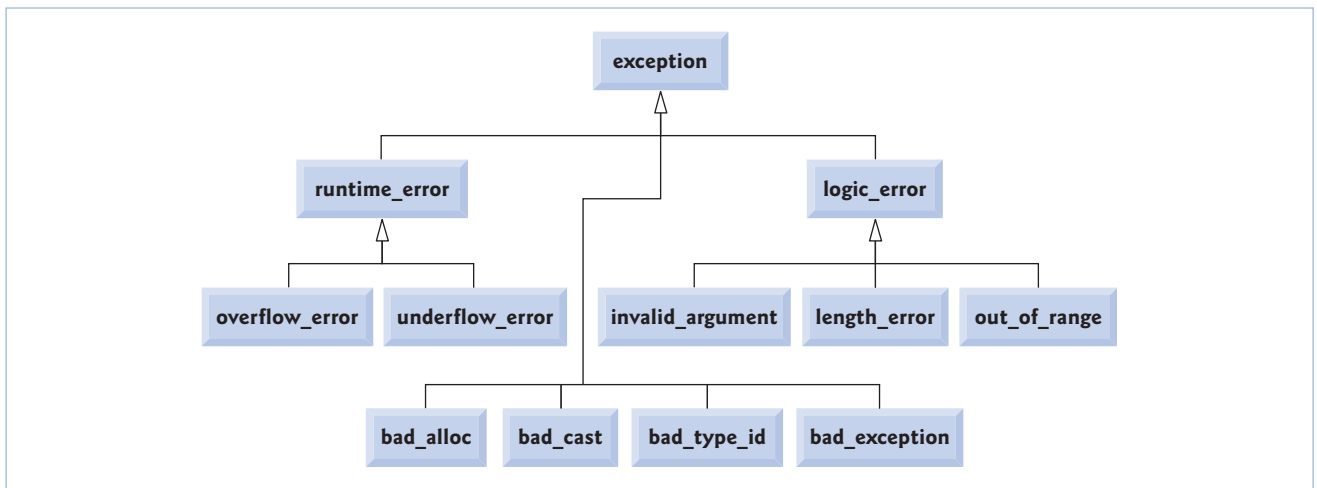


Figura 16.11 Classes de exceções da biblioteca-padrão.

A classe `logic_error` é a classe básica de várias classes de exceções padrão que indicam erros na lógica do programa. Por exemplo, a classe `invalid_argument` indica que um argumento inválido foi passado para uma função. (A codificação adequada pode, naturalmente, impedir que argumentos inválidos alcancem uma função.) A classe `length_error` indica que um comprimento maior que o tamanho máximo permitido para o objeto sendo manipulado foi utilizado para esse objeto. A classe `out_of_range` indica que um valor, como um subscrito em um array, excedeu seu intervalo permitido de valores.

A classe `runtime_error`, que utilizamos brevemente na Seção 16.8, é a classe básica de várias outras classes de exceções padrão que indicam erros de tempo de execução. Por exemplo, a classe `overflow_error` descreve um **erro de overflow aritmético** (isto é, o resultado de uma operação aritmética é maior que o maior número que pode ser armazenado no computador) e a classe `underflow_error` descreve um **erro de underflow aritmético** (isto é, o resultado de uma operação aritmética é menor que o menor número que pode ser armazenado no computador).



### Erro comum de programação 16.9

As classes de exceções definidas pelo programador não precisam ser derivadas da classe `exception`. Portanto, escrever `catch ( exception anyException )` não garante a captura (catch) de todas as exceções que um programa poderia encontrar.



### Dica de prevenção de erro 16.6

Para capturar todas as exceções potencialmente lançadas em um bloco `try`, utilize `catch (...)`. Uma fraqueza com a captura de exceções dessa maneira é que o tipo da exceção capturada é desconhecido em tempo de compilação. Outra fraqueza é que, sem um parâmetro identificado, não há nenhuma maneira de referenciar o objeto exceção dentro do handler de exceção.



### Observação de engenharia de software 16.10

A hierarquia `exception` padrão é um bom ponto de partida para criar exceções. Os programadores podem construir programas que podem lançar exceções-padrão, lançar exceções derivadas das exceções-padrão ou lançar suas próprias exceções não derivadas das exceções-padrão.



### Observação de engenharia de software 16.11

Utilize `catch (...)` para realizar uma recuperação que não depende do tipo de exceção (por exemplo, liberar recursos comuns). A exceção pode ser relançada para alertar handlers `catch` envolventes mais específicos.

## 16.14 Outras técnicas de tratamento de erro

Discutimos várias maneiras de lidar com situações excepcionais antes deste capítulo. Segue um resumo dessas e de outras técnicas de tratamento de erros:

- Ignore a exceção. Se ocorrer uma exceção, o programa pode falhar como resultado da exceção não interceptada. Isso é devastador para produtos de software comerciais ou softwares de uso especial projetados para situações de missão crítica, mas é comum para softwares desenvolvidos para seus próprios propósitos ignorar muitos tipos de erros.



## Erro comum de programação 16.10

*Abortar um componente de programa devido a uma exceção não interceptada poderia deixar um recurso — como um fluxo de arquivo ou um dispositivo de E/S — em um estado em que outros programas seriam incapazes de adquiri-lo. Isso é conhecido como um ‘vazamento de recurso’.*

- Aborto o programa. Isso, naturalmente, impede um programa de executar até a conclusão e produzir resultados incorretos. Para muitos tipos de erros, isso é adequado, especialmente para erros não fatais que permitem a um programa executar até sua conclusão (levando potencialmente o programador a pensar que o programa funcionou corretamente). Essa estratégia é inadequada aos aplicativos de missão crítica. As questões de recurso também são importantes aqui. Se conseguir um recurso, o programa deve liberá-lo antes da terminação do programa.
- Configure os indicadores de erros. O problema com essa abordagem é que os programas poderiam não verificar esses indicadores de erro em todos os pontos em que os erros poderiam ser problemáticos.
- Teste a condição de erro, emita uma mensagem de erro e chame `exit` (em `<cstdlib>`) para passar um código de erro apropriado ao ambiente do programa.
- Utilize as funções `setjmp` e `longjmp`. Essas funções de biblioteca `<setjmp>` permitem ao programador especificar um salto imediato de uma chamada de função profundamente aninhada para um handler de erro. Sem utilizar `setjmp` ou `longjmp`, um programa deve executar vários retornos para sair das chamadas de função profundamente aninhadas. As funções `setjmp` e `longjmp` são perigosas, porque elas desempilham a pilha sem chamar os destrutores para objetos automáticos. Isso pode levar a problemas sérios.
- Certos tipos específicos de erros têm capacidades dedicadas para tratá-los. Por exemplo, quando o operador `new` não consegue alocar memória, ele pode fazer com que uma função `new_handler` execute para tratar o erro. Essa função pode ser personalizada fornecendo um nome de função como o argumento para `set_new_handler`, como discutimos na Seção 16.11.

## 16.15 Síntese

Neste capítulo, você aprendeu a utilizar o tratamento de exceções para lidar com erros em um programa. Você aprendeu que o tratamento de exceções permite aos programadores remover código de tratamento de erro a partir da ‘linha principal’ da execução do programa. Demonstramos o tratamento de exceções no contexto de um exemplo de divisão por zero. Mostramos também como utilizar blocos `try` para incluir código que pode lançar uma exceção e como utilizar os handlers `catch` para lidar com as possíveis exceções. Você aprendeu a lançar e relançar exceções e a tratar as exceções que ocorrem em construtores. O capítulo continuou com discussões sobre o processamento de falhas de `new`, a alocação dinâmica de memória com classe `auto_ptr` e a hierarquia de exceções da biblioteca-padrão. No próximo capítulo, discutiremos o processamento de arquivo, incluindo a maneira como os dados persistentes são armazenados e como manipulá-los.

### Resumo

- Uma exceção é uma indicação de um problema que ocorre durante a execução de um programa.
- O tratamento de exceções permite aos programadores criar programas que podem resolver problemas que ocorrem em tempo de execução — permitindo muitas vezes aos programas continuar executando como se nenhum problema tivesse sido encontrado. Problemas mais graves podem exigir que um programa os notifique ao usuário antes de terminar de uma maneira controlada.
- O tratamento de exceções permite aos programadores remover da ‘linha principal’ de execução do programa o código de tratamento de erro, aprimorando a clareza do programa e destacando sua capacidade de modificação.
- O C++ utiliza o modelo de terminação do tratamento de exceções.
- Um bloco `try` consiste na palavra-chave `try` seguida por chaves `{ }` que definem um bloco de código em que as exceções poderiam ocorrer. O bloco `try` inclui instruções que poderiam produzir exceções e instruções que não devem executar se elas ocorrerem.
- Pelo menos um handler `catch` deve vir imediatamente depois de um bloco `try`. Cada handler `catch` especifica um parâmetro de exceção que representa o tipo de exceção que o handler `catch` pode processar.
- Se um parâmetro de exceção inclui um nome de parâmetro opcional, o handler `catch` pode utilizar esse nome de parâmetro para interagir com um objeto de exceção capturado.
- O ponto no programa em que uma exceção ocorre é chamado de ponto de lançamento.
- Se ocorrer uma exceção em um bloco `try`, ele expira e o controle do programa é transferido para o primeiro `catch` em que o tipo do parâmetro de exceção corresponde àquele da exceção lançada.
- Quando um bloco `try` termina, as variáveis locais definidas no bloco saem de escopo.

- Quando um bloco `try` termina por causa de uma exceção, o programa procura o primeiro handler `catch` que pode processar o tipo de exceção que ocorreu. O programa localiza o `catch` correspondente comparando o tipo de exceção lançado com o tipo de parâmetro de exceção de cada `catch` até encontrar uma correspondência. Ocorre uma correspondência se os tipos forem idênticos ou se o tipo da exceção lançada for uma classe derivada do tipo de parâmetro de exceção. Quando ocorrer uma correspondência, o código contido no handler `catch` correspondente executará.
- Quando um handler `catch` termina o processamento, o parâmetro `catch` e as variáveis locais definidas dentro do handler `catch` saem de escopo. Todos os handlers `catch` restantes que corresponderem ao bloco `try` serão ignorados e a execução retomarà na primeira linha de código depois da sequência `try...catch`.
- Se não ocorrer nenhuma exceção em um bloco `try`, o programa ignora o(s) handler(s) `catch` para esse bloco. A execução do programa retoma com a próxima instrução depois da sequência `try...catch`.
- Se uma exceção que ocorre em um bloco `try` não tiver nenhum handler `catch` correspondente, ou se ocorrer uma exceção em uma instrução que não está em um bloco `try`, a função que contém a instrução termina imediatamente e o programa tenta localizar um bloco `try` envolvente na função chamadora. Esse processo é chamado desempilhamento.
- O tratamento de exceções é para os erros síncronos, que ocorrem quando uma instrução executa.
- O tratamento de exceções não é projetado para processar erros associados com eventos assíncronos, que ocorrem em paralelo com o, e independente do, fluxo de controle do programa.
- Para lançar uma exceção, utilize a palavra-chave `throw` seguida por um operando que representa o tipo de exceção a lançar. Normalmente, uma instrução `throw` especifica um operando.
- O operando de um `throw` pode ser de qualquer tipo.
- O handler de exceção pode adiar o tratamento de exceções (ou talvez uma parte dele) para outro handler de exceção. Em qualquer caso, o handler alcança isso relançando a exceção.
- Exemplos comuns de exceções são subscritos de array fora do intervalo, overflow aritmético, divisão por zero, parâmetros de função inválidos e alocações de memória malsucedidas.
- A classe `exception` é a classe básica padrão do C++ para exceções. A classe `exception` fornece uma função virtual `what` que retorna uma mensagem de erro apropriada e pode ser sobrescrita em classes derivadas.
- Uma especificação de exceção opcional enumera uma lista de exceções que uma função pode lançar. Uma função pode lançar somente exceções dos tipos indicados pela especificação de exceção ou exceções de qualquer tipo derivado desses tipos. Se a função lança uma exceção que não pertence a um tipo especificado, a função `unexpected` é chamada e o programa termina normalmente.
- Uma função sem especificação de exceção pode lançar qualquer exceção. A especificação de exceção vazia `throw()` indica que uma função não lança exceções. Se uma função com uma especificação de exceção vazia tenta lançar uma exceção, a função `unexpected` é invocada.
- A função `unexpected` chama a função registrada junto à função `set_unexpected`. Se nenhuma função foi registrada dessa maneira, a função `terminate` é chamada por padrão.
- A função `set_terminate` pode especificar a função a invocar quando `terminate` for chamada. Caso contrário, `terminate` chama `abort`, que termina o programa sem chamar os destrutores de objetos que são declarados `static` e `auto`.
- As funções `set_terminate` e `set_unexpected` retornam um ponteiro à última função chamada por `terminate` e `unexpected`, respectivamente (0, na primeira vez em que cada uma é chamada). Isso permite ao programador salvar o ponteiro de função para que ele possa ser restaurado mais tarde.
- As funções `set_terminate` e `set_unexpected` aceitam como argumentos ponteiros de funções com tipo de retorno `void` e sem argumentos.
- Se uma função de terminação definida pelo programador não fecha um programa, a função `abort` será chamada depois que a função definida de terminação pelo programador completar a execução.
- Desempilhar a pilha de chamadas de função significa que a função em que a exceção não foi capturada termina, todas as variáveis locais nessa função são destruídas e o controle retorna à instrução que originalmente invocou essa função.
- A classe `runtime_error` (definida no cabeçalho `<stdexcept>`) é a classe básica padrão do C++ para representar erros de tempo de execução.
- As exceções lançadas por um construtor fazem com que os destrutores sejam chamados por qualquer objeto construído como parte do objeto sendo construído antes de a exceção ser lançada.
- Os destrutores são chamados para cada objeto automático construído em um bloco `try` antes de uma exceção ser lançada.
- O desempilhamento é concluído antes de um handler de exceção começar a executar.
- Se um destrutor invocado como resultado do desempilhamento lançar uma exceção, `terminate` é chamada.
- Se um objeto tiver objetos-membro e se uma exceção for lançada antes de o objeto externo ser completamente construído, então os destrutores serão executados para os objetos-membro que foram construídos antes da ocorrência da exceção.

- Se um array de objetos foi parcialmente construído durante a ocorrência de uma exceção, somente os destrutores para os objetos de elemento do array construídos serão chamados.
- Quando uma exceção é lançada do construtor de um objeto que é criado em uma expressão `new`, a memória dinamicamente alocada desse objeto é liberada.
- Se um handler `catch` captura um ponteiro ou referência para um objeto de exceção de um tipo de classe básica, ele também pode capturar um ponteiro ou referência para todos os objetos de classes derivadas publicamente a partir dessa classe básica — isso permite o processamento polimórfico de erros relacionados.
- O documento C++ padrão especifica que, quando o operador `new` falha, ele lança uma exceção `bad_alloc` (definida no arquivo de cabeçalho `<new>`).
- A função `set_new_handler` aceita como seu argumento um ponteiro para uma função que não aceita argumentos e retorna `void`. Esse ponteiro aponta para a função que será chamada se `new` falhar.
- Uma vez que `set_new_handler` registra um handler `new` no programa, o operador `new` não lança `bad_alloc` em caso de falha; em vez disso, ele transfere o tratamento de erro para a função handler de `new`.
- Se `new` aloca a memória com sucesso, ele retorna um ponteiro para essa memória.
- Se ocorrer uma exceção depois da alocação de memória bem-sucedida, mas antes de a instrução `delete` executar, um vazamento de memória pode ocorrer.
- A C++ Standard Library fornece o template da classe `auto_ptr` para lidar com vazamentos de memória.
- Um objeto da classe `auto_ptr` mantém um ponteiro para a memória dinamicamente alocada. O destrutor de um objeto `auto_ptr` realiza uma operação `delete` no membro de dados do ponteiro `auto_ptr`.
- O template da classe `auto_ptr` fornece os operadores sobrecarregados `*` e `->` para que um objeto `auto_ptr` possa ser utilizado da mesma maneira que uma variável de ponteiro regular. Um `auto_ptr` também transfere a posse da memória dinâmica que ele gerencia via seu construtor de cópia e operador de atribuição sobrecarregado.
- A C++ Standard Library inclui uma hierarquia de classes de exceções. Essa hierarquia é comandada pela classe básica `exception`.
- As classes derivadas imediatas da classe básica `exception` incluem `runtime_error` e `logic_error` (ambas definidas no cabeçalho `<stdexcept>`), cada uma das quais possuindo várias classes derivadas.
- Vários operadores lançam exceções-padrão — o operador `new` lança `bad_alloc`, o operador `dynamic_cast` lança `bad_cast` e o operador `typeid` lança `bad_typeid`.
- Incluir `bad_exception` na lista `throw` de uma função significa que, se ocorrer uma exceção inesperada, a função `unexpected` pode lançar uma `bad_exception` em vez de terminar a execução do programa ou chamar outra função especificada por `set_unexpected`.

## Terminologia

|                                                       |                                              |                                                                     |
|-------------------------------------------------------|----------------------------------------------|---------------------------------------------------------------------|
| <code>&lt;exception&gt;</code> , arquivo de cabeçalho | especificação de exceção                     | ponto <code>throw</code> (ponto de lançamento)                      |
| <code>&lt;memory&gt;</code> , arquivo de cabeçalho    | especificação de exceção vazia               | programas tolerantes a falha                                        |
| <code>&lt;stdexcept&gt;</code> , arquivo de cabeçalho | evento assíncrono                            | relançar uma exceção                                                |
| <code>abort</code> , função                           | exceção                                      | <code>runtime_error</code> , exceção                                |
| aplicativo robusto                                    | <code>exception</code> , classe              | <code>set_new_handler</code> , função                               |
| <code>auto_ptr</code> , template de classe            | handler de exceção                           | <code>set_terminate</code> , função                                 |
| <code>bad_alloc</code> , exceção                      | handler de falha de <code>new</code>         | <code>set_unexpected</code> , função                                |
| <code>bad_cast</code> , exceção                       | <code>invalid_argument</code> , exceção      | terminação, modelo de tratamento de exceções                        |
| <code>bad_exception</code> , exceção                  | lançar uma exceção                           | <code>terminate</code> , função                                     |
| <code>bad_typeid</code> , exceção                     | lançar uma exceção inesperada                | <code>throw</code> sem argumentos                                   |
| capturar todas as exceções                            | <code>length_error</code> , exceção          | <code>throw</code> , palavra-chave                                  |
| capturar uma exceção                                  | lista <code>throw</code>                     | tratamento de exceções                                              |
| <code>catch(...)</code>                               | <code>logic_error</code> , exceção           | tratar uma exceção                                                  |
| <code>catch</code> , handler                          | modelo de retomada do tratamento de exceções | <code>try</code> , bloco                                            |
| <code>catch</code> , palavra-chave                    | <code>nothrow</code> , objeto                | <code>try</code> , palavra-chave                                    |
| desempilhamento de pilha                              | objeto de exceção                            | <code>underflow_error</code> , exceção                              |
| erro de overflow aritmético                           | <code>out_of_range</code> , exceção          | <code>unexpected</code> , função                                    |
| erro de underflow aritmético                          | <code>overflow_error</code> , exceção        | <code>what</code> , função virtual da classe <code>exception</code> |
| erros síncronos                                       | parâmetro de exceção                         |                                                                     |

## Exercícios de revisão

- 16.1** Liste cinco exemplos de exceções comuns.
- 16.2** Apresente várias razões pelas quais as técnicas de tratamento de exceções não devem ser utilizadas para o controle convencional de programa.
- 16.3** Por que as exceções são apropriadas para lidar com erros produzidos por funções de biblioteca?
- 16.4** O que é um ‘vazamento de recurso’?
- 16.5** Se nenhuma exceção é lançada em um bloco `try`, para onde o controle prossegue depois que o bloco `try` termina de executar?
- 16.6** O que acontece se uma exceção é lançada fora de um bloco `try`?
- 16.7** Apresente uma vantagem-chave e uma desvantagem-chave de utilizar `catch(...)`.
- 16.8** O que acontece se nenhum handler `catch` corresponder ao tipo de um objeto lançado?
- 16.9** O que acontece se vários handlers corresponderem ao tipo do objeto lançado?
- 16.10** Por que um programador especificaria um tipo de classe básica como o tipo de um handler `catch` e, então, os objetos `throw` de tipos de classe derivada?
- 16.11** Suponha que um handler `catch` com uma correspondência precisa a um tipo de objeto de exceção esteja disponível. Em que circunstâncias um handler diferente poderia ser executado para objetos de exceção desse tipo?
- 16.12** O lançamento de uma exceção deve causar a terminação do programa?
- 16.13** O que acontece quando um handler `catch` lança uma exceção?
- 16.14** O que a instrução `throw` faz?
- 16.15** Como o programador restringe os tipos de exceção que uma função pode lançar?
- 16.16** O que acontece se uma função lança uma exceção de um tipo não permitido pela especificação de exceção da função?
- 16.17** O que acontece para os objetos automáticos que foram construídos em um bloco `try` quando esse bloco lança uma exceção?

## Respostas dos exercícios de revisão

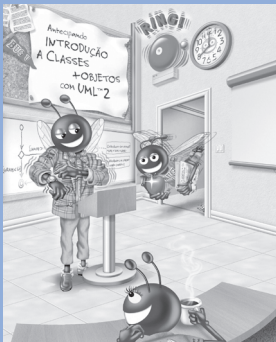
- 16.1** Memória insuficiente para satisfazer uma solicitação `new`, subscrito de array fora da linha, overflow aritmético, divisão por zero, parâmetros de função inválidos.
- 16.2** (a) O tratamento de exceções foi projetado para tratar situações que ocorrem raramente mas que frequentemente resultam na terminação do programa, portanto os escritores de compilador não são obrigados a implementar tratamento de exceções para otimizar o desempenho. (b) O fluxo de controle com estruturas de controle convencionais é geralmente mais claro e mais eficiente do que com as exceções. (c) Podem ocorrer problemas porque a pilha é desempilhada quando uma exceção ocorre e os recursos alocados antes da exceção não podem ser liberados. (d) As exceções ‘adicionais’ tornam mais difícil para o programador tratar o maior número de casos de exceção.
- 16.3** É improvável que uma função de biblioteca realize processamento de erro que atenda às necessidades únicas de todos os usuários.
- 16.4** Um programa que termina bruscamente poderia deixar um recurso em um estado em que outros programas não seriam capazes de adquiri-lo, ou o próprio programa talvez não fosse capaz de readquirir um recurso que ‘vazou’.
- 16.5** Os handlers de exceção (nos handlers `catch`) para esse bloco `try` são ignorados e o programa retoma a execução depois do último bloco `catch`.
- 16.6** Uma exceção lançada fora de um bloco `try` produz uma chamada para `terminate`.
- 16.7** A forma `catch(...)` captura qualquer tipo de exceção lançada em um bloco `try`. Uma vantagem é que todas as possíveis exceções serão capturadas. Uma desvantagem é que o `catch` não tem parâmetros, portanto, não pode referenciar informações no objeto lançado e não pode saber a causa da exceção.
- 16.8** Isso faz com que a procura por uma correspondência continue no próximo bloco `try` envolvente, se houver algum. Como esse processo continua, talvez, por fim, seja determinado que não há nenhum handler no programa que corresponda ao tipo do objeto lançado; nesse caso, é chamado `terminate`, que, por padrão, chama `abort`. Uma função `terminate` alternativa pode ser fornecida como um argumento para `set_terminate`.
- 16.9** O primeiro handler de exceção correspondente depois do bloco `try` é executado.
- 16.10** Essa é uma maneira elegante de capturar exceções relacionadas.
- 16.11** Um handler de classe básica capturaria objetos de todos os tipos de classe derivada.



- 16.12** Não, mas ele termina o bloco em que a exceção é lançada.
- 16.13** A exceção será processada por um handler `catch` (se um existir) associado com o bloco `try` (se um existir), incluindo o handler `catch` que causou a exceção.
- 16.14** Ele relança a exceção se ela aparecer em um handler `catch`; caso contrário, a função `unexpected` é chamada.
- 16.15** Fornecendo uma listagem de especificação de exceção dos tipos de exceção que a função pode lançar.
- 16.16** A função `unexpected` é chamada.
- 16.17** O bloco `try` expira, fazendo com que os destrutores sejam chamados para cada um desses objetos.

## Exercícios

- 16.18** Liste várias condições excepcionais que ocorreram por todo este texto. Liste tantas condições excepcionais adicionais que puder. Para cada uma dessas exceções, descreva brevemente como um programa trataria normalmente a exceção, utilizando as técnicas de tratamento de exceções discutidas neste capítulo. Algumas exceções típicas são a divisão por zero, o overflow aritmético, o subscrito de array fora da linha, o esgotamento do armazenamento livre etc.
- 16.19** Sob que circunstâncias o programador não forneceria um nome de parâmetro ao definir o tipo do objeto que será capturado por um handler?
- 16.20** Um programa contém a instrução  
`throw;`  
 Onde você normalmente esperaria localizar essa instrução? E se essa instrução aparecer em uma parte diferente do programa?
- 16.21** Compare e contraste o tratamento de exceções com os vários outros esquemas de processamento de erro discutidos no texto.
- 16.22** Por que as exceções não devem ser utilizadas como uma forma alternativa de controle de programa?
- 16.23** Descreva uma técnica para tratar exceções relacionadas.
- 16.24** Até este capítulo, descobrimos que é inconveniente lidar com erros detectados por construtores. O tratamento de exceções fornece um meio melhor de tratar esses erros. Considere um construtor para uma classe `String`. O construtor utiliza `new` para obter espaço de armazenamento livre. Suponha que `new` falhe. Mostre como você lidaria com isso sem o tratamento de exceções. Discuta as questões-chave. Mostre como você lidaria com esse esgotamento de memória utilizando o tratamento de exceções. Explique por que a abordagem de tratamento de exceções é superior.
- 16.25** Suponha que um programa lance uma exceção e o handler de exceção apropriado comece a executar. Agora suponha que o próprio handler de exceção lance a mesma exceção. Isso cria a recursão infinita? Escreva um programa para verificar sua observação.
- 16.26** Utilize herança para criar várias classes derivadas de `runtime_error`. Então mostre que um handler `catch` que especifica a classe básica pode capturar exceções de classe derivada.
- 16.27** Escreva uma expressão condicional que retorna um `double` ou um `int`. Forneça um handler `int catch` e um handler `double catch`. Mostre que apenas o handler `double catch` executa, independentemente de um `int` ou um `double` ser retornado.
- 16.28** Escreva um programa que gera e trata uma exceção de esgotamento de memória. Seu programa deve fazer um loop em uma solicitação para criar memória dinâmica por meio do operador `new`.
- 16.29** Escreva um programa que ilustra que todos os destrutores de objetos construídos em um bloco são chamados antes de uma exceção ser lançada a partir desse bloco.
- 16.30** Escreva um programa que ilustra que os destrutores de objeto-membro são chamados somente para aqueles objetos-membro construídos antes da ocorrência de uma exceção.
- 16.31** Escreva um programa que demonstra vários tipos de exceção sendo capturados com o handler de exceção `catch(...)`.
- 16.32** Escreva um programa que ilustra que a ordem de handlers de exceção é importante. O primeiro handler correspondente é o que executa. Tente compilar e executar o programa de duas maneiras diferentes para mostrar que os dois handlers diferentes executam com dois efeitos diferentes.
- 16.33** Escreva um programa que mostra um construtor passando informações sobre uma falha de construtor para um handler de exceção após um bloco `try`.
- 16.34** Escreva um programa que ilustra como relançar uma exceção.
- 16.35** Escreva um programa que ilustra que uma função com seu próprio bloco `try` não tem de capturar cada possível erro gerado dentro do `try`. Algumas exceções podem ser transferidas para, e tratadas em, escopos externos.
- 16.36** Escreva um programa que lança uma exceção a partir de uma função profundamente aninhada e ainda faz o handler `catch` que se segue ao bloco `try` envolvendo a cadeia de chamadas capturar a exceção.



*Li parte dele até o fim.*  
Samuel Goldwyn

*Só posso supor que um documento marcado como 'Não Arquivar' está arquivado como 'Não Arquivar'.*

Senador Frank Church  
Audiência do Subcomitê de  
Inteligência do Senado Norte-  
americano, 1975

*Uma vasta memória não faz um filósofo, assim como um dicionário não pode ser chamado de gramática.*  
John Henry, Cardeal Newman

## Processamento de arquivo

### OBJETIVOS

Neste capítulo, você aprenderá:

- Como criar, ler, gravar e atualizar arquivos.
- Processamento de arquivo seqüencial.
- Processamento de arquivo de acesso aleatório.
- Como utilizar operações de E/S não formatada de alto desempenho.
- As diferenças entre processamento de arquivo de dados formatados e dados brutos.
- A construir um programa de processamento de transação utilizando processamento de arquivo de acesso aleatório.

- 17.1** Introdução
- 17.2** Hierarquia de dados
- 17.3** Arquivos e fluxos
- 17.4** Criando um arquivo seqüencial
- 17.5** Lendo dados de um arquivo seqüencial
- 17.6** Atualizando arquivos seqüenciais
- 17.7** Arquivos de acesso aleatório
- 17.8** Criando um arquivo de acesso aleatório
- 17.9** Gravando dados aleatoriamente em um arquivo de acesso aleatório
- 17.10** Lendo de um arquivo de acesso aleatório seqüencialmente
- 17.11** Estudo de caso: um programa de processamento de transação
- 17.12** Entrada/saída de objetos
- 17.13** Síntese

Resumo | Terminologia | Exercícios de revisão | Respostas dos exercícios de revisão | Exercícios

## 17.1 Introdução

O armazenamento de dados em variáveis e arrays é temporário. Os **arquivos** são utilizados para **persistência de dados** — retenção permanente de grandes quantidades de dados. Os computadores armazenam arquivos em **dispositivos de armazenamento secundários**, como discos magnéticos, discos ópticos e fitas. Neste capítulo, explicamos como construir programas em C++ que criam, atualizam e processam arquivos de dados. Consideramos tanto os arquivos seqüenciais como os arquivos de acesso aleatório. Comparamos processamento de arquivo de dados formatados e processamento de arquivo de dados brutos. Examinamos técnicas para entrada a partir de e saída de dados para fluxos de string em vez de arquivos no Capítulo 18.

## 17.2 Hierarquia de dados

Em última instância, todos os itens de dados que computadores digitais processam são reduzidos a combinações de zeros e uns. Isso ocorre porque é simples e econômico construir dispositivos eletrônicos que podem assumir dois estados estáveis — um estado representa 0 e o outro representa 1. É notável que as impressionantes funções realizadas pelos computadores envolvam apenas as manipulações mais fundamentais de 0s e 1s.

O menor item de dados que os computadores suportam é chamado de **bit** (abreviação de ‘*binary digit*’ — um dígito que pode assumir um de dois valores). Cada item de dados, ou bit, pode assumir o valor 0 ou o valor 1. Os circuitos de computador realizam várias manipulações de bits simples como examinar o valor de um bit, configurar o valor de um bit e inverter um bit (de 1 para 0 ou de 0 para 1).

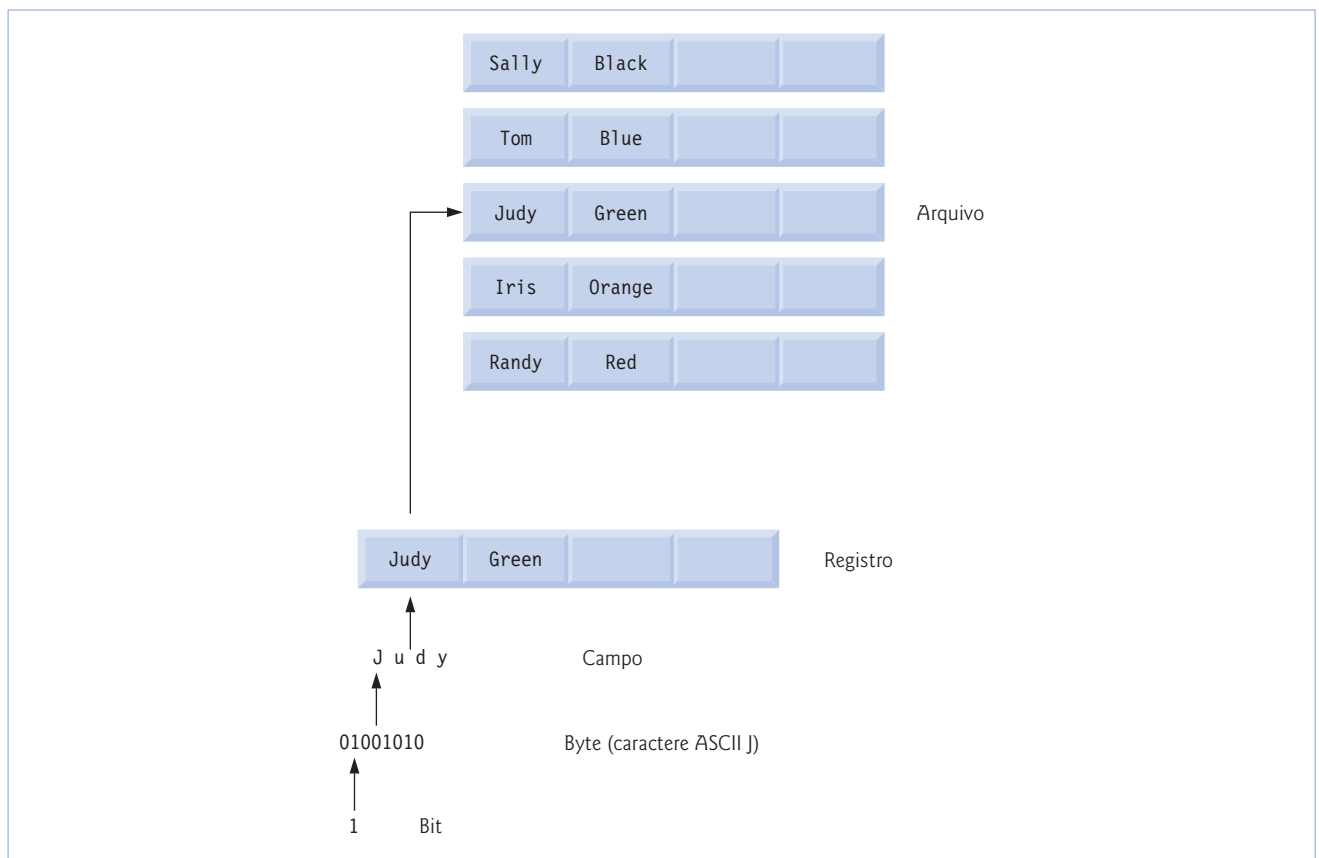
A programação com dados na forma de baixo nível de bits é complicada. É preferível programar com dados em formas como **dígitos decimais** (isto é, 0, 1, 2, 3, 4, 5, 6, 7, 8 e 9), **letras** (isto é, de A a Z e de a a z) e **símbolos especiais** (isto é, \$, @, %, &, \*, (, ), -, +, “, “, :, / e muitos outros). Dígitos, letras e símbolos especiais são referidos como **caracteres**. O conjunto de todos os caracteres utilizados para escrever programas e representar itens de dados em um computador particular é chamado de **conjunto de caracteres** desse computador. Como os computadores podem processar somente 1s e 0s, cada caractere no conjunto de caracteres de um computador é representado como um padrão de 1s e 0s. **Bytes** são compostos de oito bits. Os programadores criam programas e itens de dados com caracteres; os computadores manipulam e processam esses caracteres como padrões de bits. Por exemplo, o C++ fornece o tipo de dados char. Cada char ocupa um byte de memória. O C++ também fornece o tipo de dados wchar\_t, que pode ocupar mais de um byte (para suportar conjuntos de caracteres maiores, como o **conjunto de caracteres Unicode**<sup>®</sup>). Para obter informações adicionais sobre Unicode<sup>®</sup>, visite [www.unicode.org](http://www.unicode.org).

Assim como os caracteres são compostos de bits, os **campos** são compostos de caracteres. Um campo é um grupo de caracteres que transmite algum significado. Por exemplo, um campo consistindo em letras maiúsculas e minúsculas pode representar o nome de uma pessoa.

Os itens de dados processados por computadores formam uma **hierarquia de dados** (Figura 17.1), em que os itens de dados se tornam maiores e mais complexos em estrutura quando avançamos de bits, para caracteres, campos e grupos de dados maiores.

Em geral, um **registro** (que pode ser representado como uma class em C++) é composto de vários campos (chamados membros de dados em C++). Em um sistema de folha de pagamento, por exemplo, um registro para um empregado particular poderia incluir os seguintes campos:

1. Número de identificação do empregado
2. Nome



**Figura 17.1** Hierarquia de dados.

3. Endereço
4. Salário-hora
5. Número de isenções reivindicadas
6. Lucros no ano até a data atual
7. Total de impostos retidos

Portanto, um registro é um grupo de campos relacionados. No exemplo anterior, cada campo é associado com o mesmo empregado. Um arquivo é um grupo de registros relacionados.<sup>1</sup> O arquivo de folha de pagamento de uma empresa normalmente contém um registro de cada empregado. Portanto, o arquivo de folha de pagamento de uma pequena empresa poderia conter apenas 22 registros, enquanto o de uma grande empresa poderia conter 100.000 registros. Não é incomum uma empresa ter muitos arquivos, alguns contendo milhões, ou mesmo bilhões, de caracteres de informação.

Para facilitar a recuperação de registros específicos de um arquivo, pelo menos um campo em cada registro é escolhido como uma **chave de registro**. Uma chave de registro identifica um registro como pertencendo a uma pessoa ou entidade particular e distingue esse registro de todos os outros. No registro de folha de pagamento descrito previamente, o número de identificação de empregado normalmente seria escolhido como a chave de registro.

Há muitas maneiras de organizar registros em um arquivo. Um tipo de organização comum é chamado **arquivo seqüencial**, em que os registros são normalmente armazenados em ordem por um campo de chave de registro. Em um arquivo de folha de pagamento, os registros normalmente são ordenados pelo número de identificação do empregado. O primeiro registro de empregado no arquivo contém o menor número de identificação de empregado, e os registros subsequentes contêm números cada vez mais altos.

A maioria dos negócios utiliza muitos arquivos diferentes para armazenar dados. Por exemplo, uma empresa poderia ter arquivos de folha de pagamento, arquivos de contas a receber (listando dinheiro devido por clientes), arquivo de contas a pagar (listando dinheiro devido a fornecedores), arquivos de inventário (listando fatos sobre todos os itens manipulados pelo negócio) e muitos outros tipos de

<sup>1</sup> Geralmente, um arquivo pode conter dados arbitrários em formatos arbitrários. Em alguns sistemas operacionais, um arquivo é visto como nada mais do que uma coleção de bytes. Em sistemas operacionais assim, qualquer organização dos bytes em um arquivo (como a organização dos dados em registros) é uma visão criada pelo programador de aplicativo.

arquivo. Um grupo de arquivos relacionados costuma ser armazenado em um **banco de dados**. Uma coleção de programas projetados para criar e gerenciar bancos de dados é chamada de **sistema de gerenciamento de bancos de dados** (*database management system – DBMS*).

### 17.3 Arquivos e fluxos

O C++ visualiza cada arquivo como uma sequência de bytes (Figura 17.2). Cada arquivo termina com um **marcador de fim do arquivo** ou em um número específico de byte registrado em uma estrutura de dados administrativa mantida pelo sistema. Quando um arquivo é *aberto*, um objeto é criado e um fluxo é associado com o objeto. No Capítulo 15 vimos que os objetos `cin`, `cout`, `cerr` e `clog` são criados quando `<iostream>` é incluído. Os fluxos associados com esses objetos fornecem canais de comunicação entre um programa e um arquivo ou dispositivo particular. Por exemplo, o objeto `cin` (objeto de fluxo de entrada-padrão) permite a um programa inserir dados a partir do teclado ou outros dispositivos, o objeto `cout` (objeto de fluxo de saída-padrão) permite a um programa gerar saída de dados para a tela ou outros dispositivos, e os objetos `cerr` e `clog` (objetos de fluxo de erro-padrão) permitem a um programa gerar saída de mensagens de erro para a tela ou outros dispositivos.

Para realizar processamento de arquivo no C++, os arquivos de cabeçalho `<iostream>` e `<fstream>` devem ser incluídos. O cabeçalho `<fstream>` inclui as definições para os templates de classe de fluxo `basic_ifstream` (para entrada de arquivo), `basic_ofstream` (para saída de arquivo) e `basic_fstream` (para entrada e saída de arquivo). Cada template de classe tem uma especialização de template predefinida que permite a E/S de `char`. Além disso, a biblioteca `fstream` fornece um conjunto de typedefs que fornecem aliases dessas especializações de template. Por exemplo, o typedef `ifstream` representa uma especialização de `basic_ifstream` que permite a entrada de `char` a partir de um arquivo. De modo semelhante, o typedef `ofstream` representa uma especialização de `basic_ofstream` que permite a saída de `char` para arquivos. Além disso, o typedef `fstream` representa uma especialização de `basic_fstream` que permite entrada de `char` a partir de arquivos e a saída de `char` para arquivos.

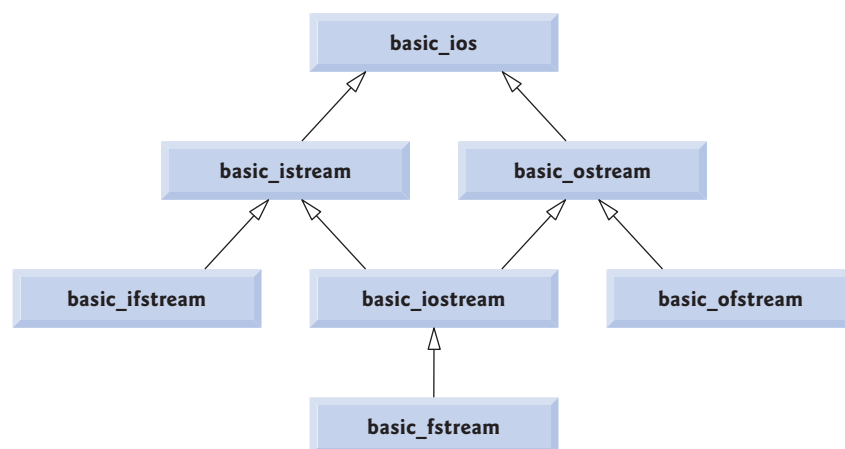
Os arquivos são abertos criando objetos dessas especializações de template de fluxo. Esses templates ‘derivam’ de templates da classe `basic_istream`, `basic_ostream` e `basic_iostream`, respectivamente. Portanto, todas as funções-membro, operadores e manipuladores que pertencem a esses templates (que descrevemos no Capítulo 15) também podem ser aplicados a fluxos de arquivo. A Figura 17.3 resume os relacionamentos de herança das classes de E/S que discutimos até agora.

### 17.4 Criando um arquivo sequencial

O C++ não impõe nenhuma estrutura a um arquivo. Assim, um conceito como o de ‘registro’ não existe em um arquivo C++. Portanto, o programador deve estruturar arquivos para atender aos requisitos do aplicativo. No exemplo a seguir, vemos como o programador impõe uma estrutura simples de registro a um arquivo.



**Figura 17.2** A visão do C++ de um arquivo de  $n$  bytes.



**Figura 17.3** Parte da hierarquia de templates de E/S de fluxo.

A Figura 17.4 cria um arquivo seqüencial simples que poderia ser utilizado em um sistema de contas a receber para ajudar a gerenciar o dinheiro devido por clientes de uma empresa. Para cada cliente, o programa obtém o número da conta do cliente, nome e saldo (isto é, a quantidade que o cliente deve à empresa por mercadoria e serviços recebidos no passado). Os dados obtidos para cada cliente constituem um registro desse cliente. O número da conta serve como a chave de registro nesse aplicativo; isto é, o programa cria e mantém o arquivo na ordem de números de conta. Esse programa assume que o usuário insere os registros na ordem de números de conta. Em um sistema abrangente de contas a receber, uma capacidade de classificação é fornecida para o usuário inserir registros em qualquer ordem — os registros então seriam classificados e gravados no arquivo.

Vamos examinar esse programa. Como declarado anteriormente, os arquivos são abertos criando os objetos `ifstream`, `ofstream` ou `fstream`. Na Figura 17.4, o arquivo deve ser aberto para saída, portanto, um objeto `ofstream` é criado. Dois argumentos são passados para o construtor do objeto — o **nome de arquivo** e o **modo file-open** (linha 19). Para um objeto `ofstream`, o modo `file-open` pode ser `ios::out` para gerar saída de dados em um arquivo ou `ios::app` para acrescentar dados ao fim de um arquivo (sem modificar quaisquer

```

1 // Figura 17.4: Fig17_04.cpp
2 // Cria um arquivo seqüencial.
3 #include <iostream>
4 using std::cerr;
5 using std::cin;
6 using std::cout;
7 using std::endl;
8 using std::ios;
9
10 #include <fstream> // fluxo de arquivo
11 using std::ofstream; // gera a saída do fluxo do arquivo
12
13 #include <cstdlib>
14 using std::exit; // sai do protótipo de função
15
16 int main()
17 {
18 // construtor ofstream abre arquivo
19 ofstream outClientFile("clients.dat", ios::out);
20
21 // fecha o programa se não conseguir criar arquivo
22 if (!outClientFile) // operador ! sobrecarregado
23 {
24 cerr << "File could not be opened" << endl;
25 exit(1);
26 } // fim do if
27
28 cout << "Enter the account, name, and balance." << endl
29 << "Enter end-of-file to end input.\n? ";
30
31 int account;
32 char name[30];
33 double balance;
34
35 // lê conta, nome e saldo a partir de cin, então coloca no arquivo
36 while (cin >> account >> name >> balance)
37 {
38 outClientFile << account << ' ' << name << ' ' << balance << endl;
39 cout << "? ";
40 } // fim do while
41
42 return 0; // destrutor ofstream fecha o arquivo
43 } // fim de main

```

**Figura 17.4** Criando um arquivo seqüencial.

(continua)

```

Enter the account, name, and balance.
Enter end-of-file to end input.
? 100 Jones 24.98
? 200 Doe 345.67
? 300 White 0.00
? 400 Stone -42.16
? 500 Rich 224.62
? ^Z

```

Figura 17.4 Criando um arquivo seqüencial.

(continuação)

dados já presentes no arquivo). Arquivos existentes abertos com o modo `ios::out` são **truncados** — todos os dados no arquivo são descartados. Se o arquivo especificado ainda não existir, então `ofstream` cria o arquivo, utilizando esse nome de arquivo.

A linha 19 cria um objeto `ofstream` chamado `outClientFile` associado com o arquivo `clients.dat` que é aberto para saída. Os argumentos `"clients.dat"` e `ios::out` são passados para o construtor `ofstream`, que abre o arquivo. Isso estabelece uma ‘linha de comunicação’ com o arquivo. Por padrão, os objetos `ofstream` são abertos para saída, então a linha 19 poderia ter executado a instrução

```
ofstream outClientFile("clients.dat");
```

para abrir `clients.dat` para saída. A Figura 17.5 lista os modos de `file-open`.



### Erro comum de programação 17.1

*Seja cuidadoso ao abrir um arquivo existente para saída (`ios::out`), especialmente quando você quiser preservar o conteúdo do arquivo, que será descartado sem avisos.*

Um objeto `ofstream` pode ser criado sem abrir um arquivo específico — um arquivo pode ser posteriormente anexado ao objeto. Por exemplo, a instrução

```
ofstream outClientFile;
```

cria um objeto `ofstream` chamado `outClientFile`. A função-membro `ofstream` **open** abre um arquivo e o anexa a um objeto `ofstream` existente como mostrado a seguir:

```
outClientFile.open("clients.dat", ios::out);
```



### Erro comum de programação 17.2

*Não abrir um arquivo antes de tentar referenciá-lo em um programa resultará em um erro.*

Depois de criar um objeto `ofstream` e tentar abri-lo, o programa testa se a operação de abertura foi ou não bem-sucedida. A instrução `if` nas linhas 22–26 utiliza a função-membro `operator!` do operador `ios` sobrecarregado para determinar se a operação `open` foi ou não bem-sucedida. A condição retorna um valor `true` se o `failbit` ou o `badbit` estiver configurado para o fluxo na operação `open`. Alguns possíveis erros são tentar abrir um arquivo inexistente para leitura, tentar abrir um arquivo para leitura ou gravação sem permissão e abrir um arquivo para gravação quando não há espaço em disco disponível.

| Modo                     | Descrição                                                                                                                                                                         |
|--------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>ios::app</code>    | Acrescenta toda saída ao fim do arquivo.                                                                                                                                          |
| <code>ios::ate</code>    | Abre um arquivo para saída e move-se para o fim do arquivo (normalmente utilizado para acrescentar dados a um arquivo). Os dados podem ser gravados em qualquer lugar do arquivo. |
| <code>ios::in</code>     | Abre um arquivo para a entrada.                                                                                                                                                   |
| <code>ios::out</code>    | Abre um arquivo para a saída.                                                                                                                                                     |
| <code>ios::trunc</code>  | Descarta o conteúdo do arquivo se ele existir (essa também é a ação-padrão de <code>ios::out</code> ).                                                                            |
| <code>ios::binary</code> | Abre um arquivo para entrada ou saída binária (isto é, não-texto).                                                                                                                |

Figura 17.5 Modos de `file-open`.



Se a condição indica uma tentativa malsucedida de abrir o arquivo, a linha 24 gera a saída da mensagem de erro "File could not be opened", e a linha 25 invoca a função `exit` para terminar o programa. O argumento para `exit` é retornado ao ambiente a partir do qual o programa foi invocado. O argumento 0 indica que o programa terminou normalmente; qualquer outro valor indica que o programa terminou por causa de um erro. O ambiente chamador (muito provavelmente o sistema operacional) utiliza o valor retornado por `exit` para responder apropriadamente ao erro.

Outra função-membro de operador `ios` sobrecarregado — `operator void *` — converte o fluxo em um ponteiro, assim ele pode ser testado como 0 (isto é, o ponteiro nulo) ou não-zero (isto é, qualquer outro valor de ponteiro). Quando um valor de ponteiro é utilizado como uma condição, o C++ converte um ponteiro nulo no valor `bool false` e converte um ponteiro não-nulo no valor `bool true`. Se o `failbit` ou o `badbit` (ver Capítulo 15) tiver sido configurado para o fluxo, 0 (`false`) é retornado. A condição na instrução `while` das linhas 36–40 invoca a função-membro `operator void *` em `cin` implicitamente. A condição permanece `true` contanto que nem `failbit` nem `badbit` tenham sido configurados para `cin`. Inserir o indicador de fim do arquivo configura o `failbit` para `cin`. A função `operator void *` pode ser utilizada para testar o fim do arquivo em um objeto de entrada em vez de chamar a função-membro `eof` explicitamente no objeto de entrada.

Se a linha 19 abra o arquivo com sucesso, o programa iniciaria o processamento dos dados. As linhas 28–29 solicitam para o usuário inserir os vários campos a cada registro ou o indicador de fim do arquivo quando a entrada de dados estiver completa. A Figura 17.6 lista as combinações de tecla para inserir o fim de arquivo em vários sistemas de computador.

A linha 36 extrai cada conjunto de dados e determina se o fim do arquivo foi inserido. Quando o fim do arquivo é encontrado ou dados inválidos são inseridos, `operator void *` retorna o ponteiro nulo (que é convertido no valor `bool false`) e a instrução `while` termina. O usuário insere o fim do arquivo para instruir o programa a não processar dados adicionais. O indicador de fim do arquivo é configurado quando o usuário insere a combinação de teclas de fim do arquivo. A instrução `while` entra em loop até que o indicador de fim do arquivo seja configurado.

A linha 38 grava um conjunto de dados no arquivo `clients.dat`, utilizando o operador de inserção de fluxo `<<` e o objeto `outClientFile` associado com o arquivo no início do programa. Os dados podem ser recuperados por um programa projetado para ler o arquivo (ver Seção 17.5). Observe que, como o arquivo criado na Figura 17.4 é simplesmente um arquivo de texto, ele pode ser visualizado por qualquer editor de textos.

Uma vez que o usuário insere o indicador de fim do arquivo, `main` termina. Isso invoca a função destrutora do objeto `outClientFile` implicitamente, o que fecha o arquivo `clients.dat`. O programador também pode fechar o objeto `ofstream` explicitamente, utilizando a função-membro `close` na instrução

```
outClientFile.close();
```



### Dica de desempenho 17.1

*Fechar arquivos explicitamente quando o programa não precisa mais referenciá-los pode reduzir o uso de recursos (especialmente se o programa continuar a executar depois de fechar os arquivos).*

Na execução de exemplo do programa da Figura 17.4, o usuário insere informações de cinco contas, e então sinaliza que a entrada de dados está completa inserindo o fim do arquivo (^Z é exibido no Microsoft Windows). Essa janela de diálogo não mostra como os registros de dados aparecem no arquivo. Para verificar se o programa criou o arquivo com sucesso, a próxima seção mostra como criar um programa que lê esse arquivo e imprime seu conteúdo.

## 17.5 Lendo dados de um arquivo seqüencial

Os arquivos armazenam dados de modo que eles possam ser recuperados para processamento quando necessário. A seção anterior demonstrou como criar um arquivo para acesso seqüencial. Nesta seção, discutimos como ler dados seqüencialmente de um arquivo.

A Figura 17.7 lê os registros do arquivo `clients.dat` que criamos utilizando o programa da Figura 17.4 e exibe o conteúdo desses registros. Criar um objeto `ifstream` abre um arquivo para entrada. O construtor `ifstream` pode receber o nome do arquivo e o modo de abertura do arquivo como argumentos. A linha 31 cria um objeto `ifstream` chamado `inClientFile` e o associa com o arquivo `clients.dat`. Os argumentos entre parênteses são passados para a função construtora `ifstream`, que abre o arquivo e estabelece uma 'linha de comunicação' com o arquivo.

| Sistema de computador | Combinação de teclas                                                   |
|-----------------------|------------------------------------------------------------------------|
| UNIX/Linux/Mac OS X   | <ctrl-d> (em uma linha isolada)                                        |
| Microsoft Windows     | <ctrl-z> (às vezes seguido pelo pressionamento de tecla <i>Enter</i> ) |
| VAX (VMS)             | <ctrl-z>                                                               |

**Figura 17.6** Combinações de tecla de fim de arquivo em vários sistemas de computador populares.



## Boa prática de programação 17.1

Abra um arquivo para entrada somente (utilizando `ios::in`) se o conteúdo do arquivo não deve ser modificado. Isso evita a modificação não intencional do conteúdo do arquivo e é um exemplo do princípio de menor privilégio.

Os objetos da classe `ifstream` são abertos para leitura por padrão. Poderíamos ter utilizado a instrução

```
ifstream inClientFile("clients.dat");
```

para abrir `clients.dat` para a entrada. Assim como com um objeto `ofstream`, um objeto `ifstream` pode ser criado sem abrir um arquivo específico, porque um arquivo pode ser posteriormente anexado a ele.

O programa utiliza a condição `!inClientFile` para determinar se o arquivo foi aberto com sucesso antes de tentar recuperar os dados do arquivo. A linha 48 lê um conjunto de dados (isto é, um registro) do arquivo. Depois que a linha anterior é executada pela primeira vez, `account` tem o valor 100, `name` tem o valor "Jones" e `balance` tem o valor 24.98. Toda vez que executar, a linha 48 lê outro registro do arquivo e transfere seus valores para as variáveis `account`, `name` e `balance`. A linha 49 exibe os registros, utilizando a função `outputLine` (linhas 55–59), que usa manipuladores de fluxo parametrizados para formatar os dados para exibição. Quando o fim de arquivo é alcançado, a chamada implícita a operador `void *` na condição `while` retorna o ponteiro nulo (que é convertido no valor `bool false`), a função destrutora `ifstream` fecha o arquivo e o programa termina.

Para recuperar dados seqüencialmente de um arquivo, os programas normalmente começam a ler a partir do início do arquivo e lêem todos os dados consecutivamente até que os dados desejados sejam encontrados. Talvez seja necessário processar o arquivo várias vezes seqüencialmente (a partir do início do arquivo) durante a execução de um programa. Tanto `istream` como `ostream` fornecem funções-membro para reposicionar o **ponteiro de posição do arquivo** (o número de bytes até o próximo byte no arquivo a ser lido ou gravado). Essas funções-membro são `seekg` ('*seek get*') para `istream` e `seekp` ('*seek put*') para `ostream`. Todo objeto `istream` tem um 'ponteiro *get*', que indica o número de bytes no arquivo a partir do qual a próxima entrada deve ocorrer, e todo objeto `ostream` tem um 'ponteiro *put*', que indica o número de bytes no arquivo em que a próxima saída deve ser colocada. A instrução

```
inClientFile.seekg(0);
```

reposiciona o ponteiro de posição de arquivo no começo do arquivo (localização 0) anexado a `inClientFile`. O argumento para `seekg` normalmente é um inteiro `long`. Um segundo argumento pode ser especificado para indicar a **direção de busca** (*seek direction*). A direção de busca pode ser `ios::beg` (o padrão) para o posicionamento em relação ao início de um fluxo, `ios::cur` para posicionamento em relação à posição atual de um fluxo, ou `ios::end` para posicionamento em relação ao fim de um fluxo. O ponteiro de posição de arquivo é um valor do tipo inteiro que especifica a localização no arquivo como um número de bytes a partir da localização inicial do arquivo (isso também é referido como o **deslocamento** a partir do início do arquivo). Alguns exemplos de posicionamento do ponteiro de posição de arquivo '*get*' são

```
// posição para o enésimo byte de fileObject (supõe ios::beg)
fileObject.seekg(n);

// posiciona n bytes para a frente em fileObject
fileObject.seekg(n, ios::cur);

// posiciona n bytes para trás a partir do fim de fileObject
fileObject.seekg(n, ios::end);

// posiciona no fim de fileObject
fileObject.seekg(0, ios::end);
```

```
1 // Figura 17.7: Fig17_07.cpp
2 // Lendo e imprimindo um arquivo seqüencial.
3 #include <iostream>
4 using std::cerr;
5 using std::cout;
6 using std::endl;
7 using std::fixed;
8 using std::ios;
9 using std::left;
10 using std::right;
11 using std::showpoint;
```

**Figura 17.7** Lendo e imprimindo um arquivo seqüencial.

(continua)

```

12
13 #include <fstream> // fluxo de arquivo
14 using std::ifstream; // insere fluxo de arquivo
15
16 #include <iomanip>
17 using std::setw;
18 using std::setprecision;
19
20 #include <string>
21 using std::string;
22
23 #include <cstdlib>
24 using std::exit; // sai do protótipo de função
25
26 void outputLine(int, const string, double); // protótipo
27
28 int main()
29 {
30 // construtor ifstream abre o arquivo
31 ifstream inClientFile("clients.dat", ios::in);
32
33 // fecha o programa se ifstream não pôde abrir o arquivo
34 if (!inClientFile)
35 {
36 cerr << "File could not be opened" << endl;
37 exit(1);
38 } // fim do if
39
40 int account;
41 char name[30];
42 double balance;
43
44 cout << left << setw(10) << "Account" << setw(13)
45 << "Name" << "Balance" << endl << fixed << showpoint;
46
47 // exibe cada registro no arquivo
48 while (inClientFile >> account >> name >> balance)
49 outputLine(account, name, balance);
50
51 return 0; // destrutor ifstream fecha o arquivo
52 } // fim de main
53
54 // exibe um registro do arquivo
55 void outputLine(int account, const string name, double balance)
56 {
57 cout << left << setw(10) << account << setw(13) << name
58 << setw(7) << setprecision(2) << right << balance << endl;
59 } // fim da função outputLine

```

| Account | Name  | Balance |
|---------|-------|---------|
| 100     | Jones | 24.98   |
| 200     | Doe   | 345.67  |
| 300     | White | 0.00    |
| 400     | Stone | -42.16  |
| 500     | Rich  | 224.62  |

**Figura 17.7** Lendo e imprimindo um arquivo seqüencial.

(continuação)

As mesmas operações podem ser realizadas utilizando a função-membro `ostream::seekp`. As funções-membro `tellg` e `tellp` são fornecidas para retornar as localizações atuais dos ponteiros `'get'` e `'put'`, respectivamente. A seguinte instrução atribui o valor do ponteiro de posição de arquivo `'get'` à variável `location` de tipo `long`:

```
location = fileObject.tellg();
```

A Figura 17.8 permite ao gerente de crédito exibir as informações de conta daqueles clientes com saldo zero (isto é, clientes que não devem nada à empresa), saldo credor (negativo) (isto é, clientes para os quais a empresa deve dinheiro) e saldo devedor (positivo) (isto é, clientes que devem dinheiro à empresa por mercadorias e serviços recebidos no passado). O programa exibe um menu e permite ao gerente de crédito inserir uma de três opções para obter informações de crédito. A opção 1 produz uma lista de contas com saldos zero. A opção 2 produz uma lista de contas com saldos credores. A opção 3 produz uma lista de contas com saldos devedores. A opção 4 termina a execução do programa. Inserir uma opção inválida exibe o prompt para inserir outra escolha.

```
1 // Figura 17.8: Fig17_08.cpp
2 // Programa de consulta de crédito.
3 #include <iostream>
4 using std::cerr;
5 using std::cin;
6 using std::cout;
7 using std::endl;
8 using std::fixed;
9 using std::ios;
10 using std::left;
11 using std::right;
12 using std::showpoint;
13
14 #include <fstream>
15 using std::ifstream;
16
17 #include <iomanip>
18 using std::setw;
19 using std::setprecision;
20
21 #include <string>
22 using std::string;
23
24 #include <cstdlib>
25 using std::exit; // sai do protótipo de função
26
27 enum RequestType { ZERO_BALANCE = 1, CREDIT_BALANCE, DEBIT_BALANCE, END };
28 int getRequest();
29 bool shouldDisplay(int, double);
30 void outputLine(int, const string, double);
31
32 int main()
33 {
34 // construtor ifstream abre o arquivo
35 ifstream inClientFile("clients.dat", ios::in);
36
37 // fecha o programa se ifstream não pôde abrir o arquivo
38 if (!inClientFile)
39 {
40 cerr << "File could not be opened" << endl;
41 exit(1);
42 } // fim do if
43 }
```

**Figura 17.8** O programa de consulta de crédito.

(continua)

```

44 int request;
45 int account;
46 char name[30];
47 double balance;
48
49 // obtém a solicitação do usuário (por exemplo, saldo zero, credor ou devedor)
50 request = getRequest();
51
52 // processa solicitação do usuário
53 while (request != END)
54 {
55 switch (request)
56 {
57 case ZERO_BALANCE:
58 cout << "\nAccounts with zero balances:\n";
59 break;
60 case CREDIT_BALANCE:
61 cout << "\nAccounts with credit balances:\n";
62 break;
63 case DEBIT_BALANCE:
64 cout << "\nAccounts with debit balances:\n";
65 break;
66 } // fim do switch
67
68 // lê a conta, nome e saldo do arquivo
69 inClientFile >> account >> name >> balance;
70
71 // exibe conteúdo do arquivo (até eof)
72 while (!inClientFile.eof())
73 {
74 // exibe o registro
75 if (shouldDisplay(request, balance))
76 outputLine(account, name, balance);
77
78 // lê a conta, o nome e o saldo do arquivo
79 inClientFile >> account >> name >> balance;
80 } // fim do while interno
81
82 inClientFile.clear(); // redefine eof para próxima entrada
83 inClientFile.seekg(0); // reposiciona no começo de arquivo
84 request = getRequest(); // obtém solicitação adicional do usuário
85 } // fim do while externo
86
87 cout << "End of run." << endl;
88 return 0; // destrutor ifstream fecha o arquivo
89 } // fim de main
90
91 // obtém a solicitação do usuário
92 int getRequest()
93 {
94 int request; // solicitação do usuário
95
96 // exibe opções de solicitação
97 cout << "\nEnter request" << endl
98 << " 1 - List accounts with zero balances" << endl
99 << " 2 - List accounts with credit balances" << endl

```

Figura 17.8 O programa de consulta de crédito.

(continua)

```

100 << " 3 - List accounts with debit balances" << endl
101 << " 4 - End of run" << fixed << showpoint;
102
103 do // entrada da solicitação do usuário
104 {
105 cout << "\n? ";
106 cin >> request;
107 } while (request < ZERO_BALANCE && request > END);
108
109 return request;
110 } // fim da função getRequest
111
112 // determina se exibe um dado registro
113 bool shouldDisplay(int type, double balance)
114 {
115 // determina se exibe saldos zero
116 if (type == ZERO_BALANCE && balance == 0)
117 return true;
118
119 // determina se exibe saldos credores
120 if (type == CREDIT_BALANCE && balance < 0)
121 return true;
122
123 // determina se exibe saldos devedores
124 if (type == DEBIT_BALANCE && balance > 0)
125 return true;
126
127 return false;
128 } // fim da função shouldDisplay
129
130 // exibe um registro do arquivo
131 void outputLine(int account, const string name, double balance)
132 {
133 cout << left << setw(10) << account << setw(13) << name
134 << setw(7) << setprecision(2) << right << balance << endl;
135 } // fim da função outputLine

```

Enter request

```

1 - List accounts with zero balances
2 - List accounts with credit balances
3 - List accounts with debit balances
4 - End of run

```

? 1

Accounts with zero balances:

```

300 White 0.00

```

Enter request

```

1 - List accounts with zero balances
2 - List accounts with credit balances
3 - List accounts with debit balances
4 - End of run

```

? 2

Accounts with credit balances:

```

400 Stone -42.16

```

Figura 17.8 O programa de consulta de crédito.

(continua)

```

Enter request
1 - List accounts with zero balances
2 - List accounts with credit balances
3 - List accounts with debit balances
4 - End of run
? 3

Accounts with debit balances:
100 Jones 24.98
200 Doe 345.67
500 Rich 224.62

Enter request
1 - List accounts with zero balances
2 - List accounts with credit balances
3 - List accounts with debit balances
4 - End of run
? 4
End of run.

```

**Figura 17.8** O programa de consulta de crédito.

(continuação)

## 17.6 Atualizando arquivos seqüenciais

Os dados formatados e gravados em um arquivo seqüencial, como mostrado na Seção 17.4, não podem ser modificados sem o risco de destruir outros dados no arquivo. Por exemplo, se o nome ‘White’ precisa ser mudado para ‘Worthington’, o nome antigo não pode ser sobrescrito sem corromper o arquivo. O registro para White foi gravado no arquivo como

```
300 White 0.00
```

Se esse registro fosse regravado começando na mesma localização do arquivo utilizando o nome mais longo, o registro seria

```
300 Worthington 0.00
```

O novo registro contém seis caracteres a mais que o registro original. Portanto, os caracteres além do segundo ‘o’ em ‘Worthington’ sobrescreveriam o início do próximo registro seqüencial no arquivo. O problema é que, no modelo de entrada/saída formatada utilizando o operador de inserção de fluxo << e o operador de extração de fluxo >>, os campos — e, portanto, os registros — podem variar de tamanho. Por exemplo, os valores 7, 14, -117, 2.074 e 27.383 são todos ints, que armazenam o mesmo número de bytes de ‘dados brutos’ internamente (em geral, quatro bytes nas atuais máquinas de 32 bits populares). Entretanto, esses inteiros tornam-se campos de diferentes tamanhos quando sua saída é gerada como texto formatado (seqüências de caractere). Portanto, o modelo de entrada/saída formatada normalmente não é utilizado para atualizar registros no local.

Essa atualização pode ser feita de uma maneira complicada. Por exemplo, para alterar o nome anterior, os registros antes de 300 White 0.00 em um arquivo seqüencial poderiam ser copiados em um novo arquivo, o registro atualizado seria então gravado no novo arquivo e os registros depois de 300 White 0.00 seriam copiados para o novo arquivo. Isso requer processar cada registro no arquivo para atualizar um registro. Mas se muitos registros estiverem sendo atualizados em uma passagem pelo arquivo, essa técnica pode ser aceitável.

## 17.7 Arquivos de acesso aleatório

Até agora, vimos como criar arquivos seqüenciais e os pesquisamos para localizar informações. Os arquivos seqüenciais são inadequados aos **aplicativos de acesso instantâneo**, em que um registro particular deve ser imediatamente localizado. Aplicativos de acesso instantâneo comuns são sistemas de reserva de passagem aérea, sistemas bancários, sistemas de ponto-de-venda, caixas automáticos e outros tipos de **sistemas de processamento de transação** que requerem acesso rápido a dados específicos. Um banco pode ter centenas de milhares (ou até milhões) de outros clientes, e ainda assim, quando um cliente utiliza um caixa automático, o programa verifica a conta desse cliente em alguns segundos ou menos para saber se há fundos suficientes. Esse tipo de acesso instantâneo é possível com **arquivos de acesso aleatório**. Os registros individuais de um arquivo de acesso aleatório podem ser acessados diretamente (e rapidamente) sem precisar pesquisar outros registros.

Como dissemos, o C++ não impõe estrutura a um arquivo. Portanto, o aplicativo que quiser utilizar arquivos de acesso aleatório deve criá-los. Uma variedade de técnicas pode ser utilizada. Talvez o método mais fácil seja impor que todos os registros em um arquivo tenham o mesmo comprimento fixo. A utilização de registros de mesmo tamanho e largura fixa facilita para um programa calcular (como uma função do tamanho do registro e da chave do registro) a localização exata de qualquer registro em relação ao início do arquivo. Logo veremos como isso facilita o acesso imediato a registros específicos, mesmo em arquivos grandes.



A Figura 17.9 ilustra a visão do C++ de um arquivo de acesso aleatório composto de registros de largura fixa (cada registro, nesse caso, tem 100 bytes de comprimento). Um arquivo de acesso aleatório é como um trem com muitos vagões do mesmo tamanho — alguns vazios e alguns com conteúdo.

Os dados podem ser inseridos em um arquivo de acesso aleatório sem destruir outros dados no arquivo. Os dados previamente armazenados também podem ser atualizados ou excluídos sem regravar o arquivo inteiro. Nas seções a seguir, explicamos como criar um arquivo de acesso aleatório, inserir dados no arquivo, ler dados tanto sequencial como aleatoriamente, atualizar dados e excluir aqueles que não são mais necessários.

## 17.8 Criando um arquivo de acesso aleatório

A função-membro `ostream::write` gera a saída de um número fixo de bytes, começando em uma localização específica da memória, para o fluxo especificado. Quando o fluxo é associado com um arquivo, a função `write` grava os dados na localização do arquivo especificada pelo ponteiro de posição de arquivo `'put'`. A função-membro `istream::read` transfere um número fixo de bytes a partir do fluxo especificado para uma área do início da memória em um endereço especificado. Se o fluxo estiver associado com um arquivo, a função `read` insere os bytes na localização do arquivo especificada pelo ponteiro de posição de arquivo `'get'`.

### Gravando bytes com a função-membro `ostream::write`

Ao gravar um inteiro `number` em um arquivo, em vez de utilizar a instrução

```
outFile << number;
```

que, para um inteiro de quatro bytes poderia imprimir apenas um dígito ou até 11 (10 dígitos mais um sinal, cada um requerendo um único byte de armazenamento), podemos utilizar a instrução

```
outFile.write(reinterpret_cast< const char * >(&number), sizeof(number));
```

que sempre grava a versão binária dos quatro bytes do inteiro (em uma máquina com inteiros de quatro bytes). A função `write` trata seu primeiro argumento como um grupo de bytes visualizando o objeto na memória como um `const char *`, que é um ponteiro para um byte (lembre-se de que um `char` é um byte). Iniciando dessa localização, a função `write` gera a saída do número de bytes especificados por seu segundo argumento — um inteiro do tipo `size_t`. Como veremos, a função `istream::read` pode ser subsequentemente utilizada para ler os quatro bytes de volta para a variável do tipo inteiro `number`.

### Convertendo entre tipos de ponteiro com o operador `reinterpret_cast`

Infelizmente, a maioria dos ponteiros que passamos para a função `write` como o primeiro argumento não é do tipo `const char *`. Para gerar a saída de outros tipos de objetos, devemos converter os ponteiros para esses objetos no tipo `const char *`; caso contrário, o compilador não compilará chamadas para a função `write`. O C++ fornece o operador `reinterpret_cast` para casos como esse em que um ponteiro de um tipo deve sofrer coerção em um tipo de ponteiro não relacionado. Você também pode utilizar esse operador de coerção para converter entre ponteiro e tipos de inteiro, e vice-versa. Sem uma `reinterpret_cast`, a instrução `write` que gera saída do inteiro `number` não compilará porque o compilador não permite que um ponteiro de tipo `int *` (o tipo retornado pela expressão `&number`) seja passado para uma função que espera um argumento do tipo `const char *` — no que diz respeito ao compilador, esses tipos são incompatíveis.

Uma `reinterpret_cast` é realizada em tempo de compilação e não muda o valor do objeto para o qual seu operando aponta. Em vez disso, ela solicita que o compilador reinterprete o operando como o tipo de alvo (especificado nos colchetes angulares que se seguem à palavra-chave `reinterpret_cast`). Na Figura 17.12, utilizamos `reinterpret_cast` para converter um ponteiro `ClientData` em um `const char *`, que reinterpreta um objeto `ClientData` como bytes a serem enviados como saída para um arquivo. Programas de processamento de arquivo de acesso aleatório raramente gravam um único campo em um arquivo. Normalmente, eles gravam um objeto de uma classe por vez, como mostramos no seguinte exemplo.



### Dica de prevenção de erro 17.1

É fácil utilizar `reinterpret_cast` para realizar manipulações perigosas que poderiam levar a erros graves de tempo de execução.



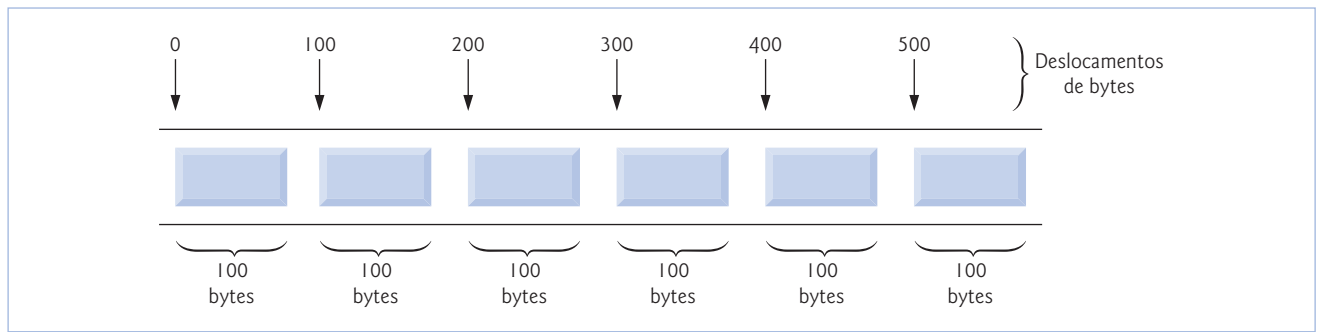
### Dica de portabilidade 17.1

O uso de `reinterpret_cast` é dependente de compilador e pode fazer com que programas se comportem diferentemente em plataformas diferentes. O operador `reinterpret_cast` não deve ser utilizado a menos que absolutamente necessário.



### Dica de portabilidade 17.2

Um programa que lê dados não formatados (gravados por `write`) deve ser compilado e executado em um sistema compatível com o programa que gravou os dados, porque sistemas diferentes podem representar os dados internos de modo diferente.



**Figura 17.9** Visão do C++ de um arquivo de acesso aleatório.

### Programa de processamento de crédito

Considere a seguinte declaração do problema:

*Crie um programa de processamento de crédito capaz de armazenar no máximo 100 registros de largura fixa para uma empresa que pode ter até 100 clientes. Cada registro deve consistir em um número de conta que atua como a chave de registro, um sobrenome, um nome e um saldo. O programa deve ser capaz de atualizar uma conta, inserir uma nova conta, excluir uma conta e inserir todos os registros de conta em um arquivo de texto formatado para impressão.*

As próximas várias seções introduzem as técnicas para criar esse programa de processamento de crédito. A Figura 17.12 ilustra como abrir um arquivo de acesso aleatório, definir o formato do registro utilizando um objeto da classe `ClientData` (figuras 17.10–17.11) e gravar dados no disco em formato binário. Esse programa inicializa todos os 100 registros do arquivo `credit.dat` com objetos vazios, utilizando a função `write`. Cada objeto vazio contém 0 para o número de conta, a string nula (representada por aspas vazias) para o nome e o sobrenome e 0.0 para o saldo. Cada registro é inicializado com a quantidade de espaço vazio em que os dados da conta serão armazenados.

Os objetos da classe `string` não têm tamanho uniforme porque utilizam memória dinamicamente alocada para acomodar strings de vários comprimentos. Esse programa deve manter os registros de largura fixa, desse modo a classe `ClientData` armazena o nome e o sobrenome do cliente em arrays `char` de largura fixa. As funções-membro `setLastName` (Figura 17.11, linhas 37–45) e `setFirstName` (Figura 17.11, linhas 54–62) copiam os caracteres de um objeto `string` para o array `char` correspondente. Considere a função `setLastName`. A linha 40 inicializa o `const char * lastNameValue` com o resultado de uma chamada à função-membro `string data`, que retorna um array contendo os caracteres da string. [Nota: Não é garantido que esse array seja terminado por um caractere nulo.] A linha 41 invoca a função-membro `string size` para obter o comprimento de `lastNameString`. A linha 42 assegura que `length` tem menos de 15 caracteres, então a linha 43 copia os caracteres `length` de `lastNameValue` no array `char lastName`. A função-membro `setFirstName` realiza os mesmos passos para o nome.

Na Figura 17.12, a linha 18 cria um objeto `ofstream` para o arquivo `credit.dat`. O segundo argumento para o construtor — `ios::binary` — indica que estamos abrindo o arquivo para saída no modo binário, que é requerido se tivermos de gravar registros de largura fixa. As linhas 31–32 fazem com que o `blankClient` grave no arquivo `credit.dat` associado com o objeto `ofstream outCredit`. Lembre-se de que o operador `sizeof` retorna o tamanho em bytes do objeto entre parênteses (ver Capítulo 8). O primeiro argumento para a função `write` na linha 31 deve ser do tipo `const char *`. Entretanto, o tipo de dados de `&blankClient` é `ClientData *`. Para converter `&blankClient` em `const char *`, a linha 31 utiliza o operador de coerção `reinterpret_cast`. Desse modo, a chamada a `write` compila sem emitir um erro de compilação.

```

1 // Figura 17.10: ClientData.h
2 // Definição da classe ClientData utilizada nas figuras 17.12–17.15.
3 #ifndef CLIENTDATA_H
4 #define CLIENTDATA_H
5
6 #include <string>
7 using std::string;
8
9 class ClientData
10 {

```

**Figura 17.10** Arquivo de cabeçalho da classe `ClientData`.

(continua)

```

11 public:
12 // construtor ClientData padrão
13 ClientData(int = 0, string = "", string = "", double = 0.0);
14
15 // funções de acesso para accountNumber
16 void setAccountNumber(int);
17 int getAccountNumber() const;
18
19 // funções de acesso para lastName
20 void setLastName(string);
21 string getLastName() const;
22
23 // funções de acesso para firstName
24 void setFirstName(string);
25 string getFirstName() const;
26
27 // funções de acesso para balance
28 void setBalance(double);
29 double getBalance() const;
30 private:
31 int accountNumber;
32 char lastName[15];
33 char firstName[10];
34 double balance;
35 }; // fim da classe ClientData
36
37 #endif

```

**Figura 17.10** Arquivo de cabeçalho da classe ClientData.

(continuação)

```

1 // Figura 17.11: ClientData.cpp
2 // Classe ClientData armazena informações de crédito do cliente.
3 #include <string>
4 using std::string;
5
6 #include "ClientData.h"
7
8 // construtor ClientData padrão
9 ClientData::ClientData(int accountNumberValue,
10 string lastNameValue, string firstNameValue, double balanceValue)
11 {
12 setAccountNumber(accountNumberValue);
13 setLastName(lastNameValue);
14 setFirstName(firstNameValue);
15 setBalance(balanceValue);
16 } // fim do construtor ClientData
17
18 // obtém o valor do número da conta
19 int ClientData::getAccountNumber() const
20 {
21 return accountNumber;
22 } // fim da função getAccountNumber
23
24 // configura o valor do número da conta

```

**Figura 17.11** A classe ClientData representa as informações de crédito de um cliente.

(continua)

```

25 void ClientData::setAccountNumber(int accountNumberValue)
26 {
27 accountNumber = accountNumberValue; // deve validar
28 } // fim da função setAccountNumber
29
30 // obtém o valor do sobrenome
31 string ClientData::getLastName() const
32 {
33 return lastName;
34 } // fim da função getLastName
35
36 // configura o valor do sobrenome
37 void ClientData::setLastName(string lastNameString)
38 {
39 // copia no máximo 15 caracteres da string para lastName
40 const char *lastNameValue = lastNameString.data();
41 int length = lastNameString.size();
42 length = (length < 15 ? length : 14);
43 strncpy(lastName, lastNameValue, length);
44 lastName[length] = '\0'; // acrescenta caractere nulo ao sobrenome
45 } // fim da função setLastName
46
47 // obtém o valor do nome
48 string ClientData::getFirstName() const
49 {
50 return firstName;
51 } // fim da função getFirstName
52
53 // configura o valor do nome
54 void ClientData::setFirstName(string firstNameString)
55 {
56 // copia no máximo 10 caracteres da string para firstName
57 const char *firstNameValue = firstNameString.data();
58 int length = firstNameString.size();
59 length = (length < 10 ? length : 9);
60 strncpy(firstName, firstNameValue, length);
61 firstName[length] = '\0'; // acrescenta o caractere nulo a firstName
62 } // fim da função setFirstName
63
64 // obtém o valor do saldo
65 double ClientData::getBalance() const
66 {
67 return balance;
68 } // fim da função getBalance
69
70 // configura o valor do saldo
71 void ClientData::setBalance(double balanceValue)
72 {
73 balance = balanceValue;
74 } // fim da função setBalance

```

**Figura 17.11** A classe ClientData representa as informações de crédito de um cliente.

(continuação)

```

1 // Figura 17.12: Fig17_12.cpp
2 // Criando um arquivo de acesso aleatório.
3 #include <iostream>
4 using std::cerr;
5 using std::endl;
6 using std::ios;
7
8 #include <fstream>
9 using std::ofstream;
10
11 #include <cstdlib>
12 using std::exit; // sai do protótipo da função
13
14 #include "ClientData.h" // definição da classe ClientData
15
16 int main()
17 {
18 ofstream outCredit("credit.dat", ios::binary);
19
20 // fecha o programa se ofstream não pôde abrir o arquivo
21 if (!outCredit)
22 {
23 cerr << "File could not be opened." << endl;
24 exit(1);
25 } // fim do if
26
27 ClientData blankClient; // construtor zera ou apaga cada membro de dados
28
29 // gera a saída de 100 registros em branco no arquivo
30 for (int i = 0; i < 100; i++)
31 outCredit.write(reinterpret_cast< const char * >(&blankClient),
32 sizeof(ClientData));
33
34 return 0;
35 } // fim de main

```

**Figura 17.12** Criando um arquivo de acesso aleatório com 100 registros em branco sequencialmente.

## 17.9 Gravando dados aleatoriamente em um arquivo de acesso aleatório

A Figura 17.13 grava os dados no arquivo `credit.dat` e utiliza a combinação de funções `fstream seekp` e `write` para armazenar dados em localizações exatas no arquivo. A função `seekp` configura o ponteiro de posição de arquivo `'put'` como uma posição específica no arquivo e, então, `write` gera saída dos dados. Observe que a linha 19 inclui o arquivo de cabeçalho `ClientData.h` definido na Figura 17.10, assim, o programa pode utilizar objetos `ClientData`.

As linhas 59–60 posicionam o ponteiro de posição de arquivo `'put'` do objeto `outCredit` na localização de byte calculada por

```
(client.getAccountNumber() - 1) * sizeof(ClientData)
```

Como o número de conta está entre 1 e 100, 1 é subtraído do número de conta ao calcular a localização de byte do registro. Portanto, para o registro 1, o ponteiro de posição no arquivo é configurado como o byte 0 do arquivo. Observe que a linha 28 utiliza o objeto `fstream outCredit` para abrir o arquivo `credit.dat` existente. O arquivo é aberto para entrada e saída no modo binário combinando os modos `file-open ios::in`, `ios::out` e `ios::binary`. Múltiplos modos `file-open` são combinados separando cada modo de abertura do próximo com o operador OU inclusivo sobre bits (`|`). Abrir o arquivo `credit.dat` existente dessa maneira assegura que esse programa pode manipular os registros gravados no arquivo pelo programa da Figura 17.12, em vez de criar o arquivo do zero. O Capítulo 22, “Bits, caracteres, strings C e structs”, discute detalhadamente o operador OU inclusivo sobre bits.

```

1 // Figura 17.13: Fig17_13.cpp
2 // Gravando em um arquivo de acesso aleatório.
3 #include <iostream>
4 using std::cerr;
5 using std::cin;
6 using std::cout;
7 using std::endl;
8 using std::ios;
9
10 #include <iomanip>
11 using std::setw;
12
13 #include <fstream>
14 using std::fstream;
15
16 #include <cstdlib>
17 using std::exit; // sai do protótipo de função
18
19 #include "ClientData.h" // definição da classe ClientData
20
21 int main()
22 {
23 int accountNumber;
24 char lastName[15];
25 char firstName[10];
26 double balance;
27
28 fstream outCredit("credit.dat", ios::in | ios::out | ios::binary);
29
30 // sai do programa se fstream não puder abrir o arquivo
31 if (!outCredit)
32 {
33 cerr << "File could not be opened." << endl;
34 exit(1);
35 } // fim do if
36
37 cout << "Enter account number (1 to 100, 0 to end input)\n? ";
38
39 // requer que usuário especifique o número da conta
40 ClientData client;
41 cin >> accountNumber;
42
43 // o usuário insere informações, que são copiadas para o arquivo
44 while (accountNumber > 0 && accountNumber <= 100)
45 {
46 // o usuário insere o sobrenome, o nome e o saldo
47 cout << "Enter lastname, firstname, balance\n? ";
48 cin >> setw(15) >> lastName;
49 cin >> setw(10) >> firstName;
50 cin >> balance;
51
52 // configura valores de accountNumber, lastName, firstName e balance
53 client.setAccountNumber(accountNumber);
54 client.setLastName(lastName);
55 client.setFirstName(firstName);
56 client.setBalance(balance);

```

**Figura 17.13** Gravando em um arquivo de acesso aleatório.

(continua)

```

57
58 // busca posição no arquivo de registro especificado pelo usuário
59 outCredit.seekp((client.getAccountNumber() - 1) *
60 sizeof(ClientData));
61
62 // grava as informações especificadas pelo usuário no arquivo
63 outCredit.write(reinterpret_cast< const char * >(&client),
64 sizeof(ClientData));
65
66 // permite ao usuário inserir outra conta
67 cout << "Enter account number\n? ";
68 cin >> accountNumber;
69 } // fim do while
70
71 return 0;
72 } // fim de main

```

```

Enter account number (1 to 100, 0 to end input)
? 37
Enter lastname, firstname, balance
? Barker Doug 0.00
Enter account number
? 29
Enter lastname, firstname, balance
? Brown Nancy -24.54
Enter account number
? 96
Enter lastname, firstname, balance
? Stone Sam 34.98
Enter account number
? 88
Enter lastname, firstname, balance
? Smith Dave 258.34
Enter account number
? 33
Enter lastname, firstname, balance
? Dunn Stacey 314.33
Enter account number
? 0

```

Figura 17.13 Gravando em um arquivo de acesso aleatório.

(continuação)

## 17.10 Lendo um arquivo de acesso aleatório seqüencialmente

Nas seções anteriores, criamos um arquivo de acesso aleatório e gravamos dados nesse arquivo. Nesta seção, desenvolvemos um programa que lê o arquivo seqüencialmente e imprime apenas aqueles registros que contêm dados. Esses programas produzem um benefício adicional. Veja se você consegue determinar qual é; nós o revelaremos no final desta seção.

A função `istream::read` insere um número especificado de bytes da posição atual no fluxo especificado em um objeto. Por exemplo, as linhas 57–58 da Figura 17.14 lêem o número de bytes especificados por `sizeof( ClientData )` a partir do arquivo associado com o objeto `ifstream inCredit` e armazenam os dados no registro `client`. Observe que a função `read` requer um primeiro argumento do tipo `char *`. Visto que `&client` é do tipo `ClientData *`, `&client` deve sofrer coerção para `char *` utilizando o operador de coerção `reinterpret_cast`. Observe que a linha 24 inclui o arquivo de cabeçalho `clientData.h` definido na Figura 17.10, assim o programa pode utilizar objetos `ClientData`.

A Figura 17.14 lê seqüencialmente cada registro no arquivo `credit.dat`, verifica todos os registros para determinar se eles contêm dados e exibe saídas formatadas para os registros contendo dados. A condição na linha 50 utiliza a função-membro `ios::eof` para determinar quando o fim de arquivo é alcançado e faz com que a execução da instrução `while` termine. Além disso, se ocorrer um erro durante a leitura do arquivo, o loop termina, porque `inCredit` é avaliado como `false`. A saída da entrada de dados do arquivo é gerada pela função



outputLine (linhas 65–72), que aceita dois argumentos — um objeto ostream e uma estrutura clientData cuja saída será realizada. O tipo de parâmetro ostream é interessante, porque qualquer objeto ostream (como cout) ou qualquer objeto de uma classe derivada de ostream (como um objeto do tipo ofstream) pode ser fornecido como o argumento. Isso significa que a mesma função pode ser utilizada, por exemplo, para realizar saída para o fluxo de saída-padrão e para um fluxo de arquivo sem escrever funções separadas.

E quanto àquele benefício adicional que prometemos? Se examinar a janela de saída, você notará que os registros são listados em ordem de classificação (por número de conta). Isso é uma consequência de como armazenamos esses registros no arquivo, utilizando as técnicas de acesso direto. Comparado à classificação por inserção que utilizamos no Capítulo 7, a classificação por meio de técnicas de acesso direto é relativamente rápida. A velocidade é alcançada tornando o arquivo suficientemente grande para armazenar cada possível registro que venha a ser criado. Isso, naturalmente, significa que o arquivo poderia estar esparsamente ocupado na maior parte do tempo, resultando em um desperdício de espaço de armazenamento. Esse é outro exemplo da relação de troca espaço-tempo: utilizando grandes quantidades de espaço, somos capazes de desenvolver um algoritmo de classificação muito mais rápido. Felizmente, a contínua redução no preço de unidades de armazenamento minimizou essa questão.

```

1 // Figura 17.14: Fig17_14.cpp
2 // Lendo um arquivo de acesso aleatório seqüencialmente.
3 #include <iostream>
4 using std::cerr;
5 using std::cout;
6 using std::endl;
7 using std::fixed;
8 using std::ios;
9 using std::left;
10 using std::right;
11 using std::showpoint;
12
13 #include <iomanip>
14 using std::setprecision;
15 using std::setw;
16
17 #include <fstream>
18 using std::ifstream;
19 using std::ofstream;
20
21 #include <cstdlib>
22 using std::exit; // sai do protótipo de função
23
24 #include "ClientData.h" // definição da classe ClientData
25
26 void outputLine(ostream&, const ClientData &); // protótipo
27
28 int main()
29 {
30 ifstream inCredit("credit.dat", ios::in);
31
32 // fecha o programa se ifstream não puder abrir o arquivo
33 if (!inCredit)
34 {
35 cerr << "File could not be opened." << endl;
36 exit(1);
37 } // fim do if
38
39 cout << left << setw(10) << "Account" << setw(16)
40 << "Last Name" << setw(11) << "First Name" << left
41 << setw(10) << right << "Balance" << endl;
42

```

**Figura 17.14** Lendo um arquivo de acesso aleatório seqüencialmente.

(continua)

```

43 ClientData client; // cria registro
44
45 // lê o primeiro registro do arquivo
46 inCredit.read(reinterpret_cast< char * >(&client),
47 sizeof(ClientData));
48
49 // lê todos os registros do arquivo
50 while (inCredit && !inCredit.eof())
51 {
52 // exibe o registro
53 if (client.getAccountNumber() != 0)
54 outputLine(cout, client);
55
56 // lê o próximo registro do arquivo
57 inCredit.read(reinterpret_cast< char * >(&client),
58 sizeof(ClientData));
59 } // fim do while
60
61 return 0;
62 } // fim de main
63
64 // exibe um único registro
65 void outputLine(ostream &output, const ClientData &record)
66 {
67 output << left << setw(10) << record.getAccountNumber()
68 << setw(16) << record.getLastName()
69 << setw(11) << record.getFirstName()
70 << setw(10) << setprecision(2) << right << fixed
71 << showpoint << record.getBalance() << endl;
72 } // fim da função outputLine

```

| Account | Last Name | First Name | Balance |
|---------|-----------|------------|---------|
| 29      | Brown     | Nancy      | -24.54  |
| 33      | Dunn      | Stacey     | 314.33  |
| 37      | Barker    | Doug       | 0.00    |
| 88      | Smith     | Dave       | 258.34  |
| 96      | Stone     | Sam        | 34.98   |

**Figura 17.14** Lendo um arquivo de acesso aleatório seqüencialmente.

(continuação)

## 17.11 Estudo de caso: um programa de processamento de transação

Agora apresentamos um programa de processamento de transação substancial (Figura 17.15) utilizando um arquivo de acesso aleatório para alcançar o processamento de acesso ‘instantâneo’. O programa mantém as informações de contas de um banco. O programa atualiza contas existentes, adiciona novas contas, exclui contas e armazena uma listagem formatada de todas as contas atuais em um arquivo de texto. Supomos que o programa da Figura 17.12 foi executado para criar o arquivo `credit.dat` e que o programa da Figura 17.13 foi executado para inserir os dados iniciais.

```

1 // Figura 17.15: Fig17_15.cpp
2 // Este programa lê um arquivo de acesso aleatório seqüencialmente, atualiza
3 // os dados previamente gravados no arquivo, cria dados a serem colocados
4 // no arquivo e exclui os dados previamente no arquivo.
5 #include <iostream>

```

**Figura 17.15** Programa de conta bancária.

(continua)

```

6 using std::cerr;
7 using std::cin;
8 using std::cout;
9 using std::endl;
10 using std::fixed;
11 using std::ios;
12 using std::left;
13 using std::right;
14 using std::showpoint;
15
16 #include <fstream>
17 using std::ofstream;
18 using std::ostream;
19 using std::fstream;
20
21 #include <iomanip>
22 using std::setw;
23 using std::setprecision;
24
25 #include <cstdlib>
26 using std::exit; // sai do protótipo de função
27
28 #include "ClientData.h" // definição da classe ClientData
29
30 int enterChoice();
31 void createTextFile(fstream&);
32 void updateRecord(fstream&);
33 void newRecord(fstream&);
34 void deleteRecord(fstream&);
35 void outputLine(ostream&, const ClientData &);
36 int getAccount(const char * const);
37
38 enum Choices { PRINT = 1, UPDATE, NEW, DELETE, END };
39
40 int main()
41 {
42 // abre o arquivo para leitura e gravação
43 fstream inOutCredit("credit.dat", ios::in | ios::out);
44
45 // fecha programa se fstream não puder abrir o arquivo
46 if (!inOutCredit)
47 {
48 cerr << "File could not be opened." << endl;
49 exit (1);
50 } // fim do if
51
52 int choice; // armazena a escolha do usuário
53
54 // permite ao usuário especificar a ação
55 while ((choice = enterChoice()) != END)
56 {
57 switch (choice)
58 {
59 case PRINT: // cria arquivo de texto a partir do arquivo de registro
60 createTextFile(inOutCredit);
61 break;

```

Figura 17.15 Programa de conta bancária.

(continua)

```

62 case UPDATE: // atualiza o registro
63 updateRecord(inOutCredit);
64 break;
65 case NEW: // cria registro
66 newRecord(inOutCredit);
67 break;
68 case DELETE: // exclui registro existente
69 deleteRecord(inOutCredit);
70 break;
71 default: // exibe erro se o usuário não selecionar uma escolha válida
72 cerr << "Incorrect choice" << endl;
73 break;
74 } // fim do switch
75
76 inOutCredit.clear(); // reinicializa indicador de fim do arquivo
77 } // fim do while
78
79 return 0;
80 } // fim de main
81
82 // permite ao usuário inserir escolhas de menu
83 int enterChoice()
84 {
85 // exibe opções disponíveis
86 cout << "\nEnter your choice" << endl
87 << "1 - store a formatted text file of accounts" << endl
88 << " called \"print.txt\" for printing" << endl
89 << "2 - update an account" << endl
90 << "3 - add a new account" << endl
91 << "4 - delete an account" << endl
92 << "5 - end program\n? ";
93
94 int menuChoice;
95 cin >> menuChoice; // insere a seleção de menu do usuário
96 return menuChoice;
97 } // fim da função enterChoice
98
99 // cria arquivo de texto formatado para impressão
100 void createTextFile(fstream &readFromFile)
101 {
102 // cria arquivo de texto
103 ofstream outPrintFile("print.txt", ios::out);
104
105 // fecha o programa se ofstream não criar arquivo
106 if (!outPrintFile)
107 {
108 cerr << "File could not be created." << endl;
109 exit(1);
110 } // fim do if
111
112 outPrintFile << left << setw(10) << "Account" << setw(16)
113 << "Last Name" << setw(11) << "First Name" << right
114 << setw(10) << "Balance" << endl;
115
116 // configura o ponteiro de posição de arquivo para o começo de readFromFile
117 readFromFile.seekg(0);

```

Figura 17.15 Programa de conta bancária.

(continua)

```

118
119 // lê o primeiro registro a partir do arquivo de registro
120 ClientData client;
121 readFromFile.read(reinterpret_cast< char * >(&client),
122 sizeof(ClientData));
123
124 // copia todos os registros a partir do arquivo de registro para um arquivo de texto
125 while (!readFromFile.eof())
126 {
127 // grava um único registro no arquivo de texto
128 if (client.getAccountNumber() != 0) // pula registros vazios
129 outputLine(outPrintFile, client);
130
131 // lê o próximo registro a partir do arquivo de registro
132 readFromFile.read(reinterpret_cast< char * >(&client),
133 sizeof(ClientData));
134 } // fim do while
135 } // fim da função createTextFile
136
137 // atualiza o saldo no registro
138 void updateRecord(fstream &updateFile)
139 {
140 // obtém o número de conta a atualizar
141 int accountNumber = getAccount("Enter account to update");
142
143 // move o ponteiro de posição de arquivo para corrigir o registro no arquivo
144 updateFile.seekg((accountNumber - 1) * sizeof(ClientData));
145
146 // lê o primeiro registro do arquivo
147 ClientData client;
148 updateFile.read(reinterpret_cast< char * >(&client),
149 sizeof(ClientData));
150
151 // atualiza o registro
152 if (client.getAccountNumber() != 0)
153 {
154 outputLine(cout, client); // exibe o registro
155
156 // solicita para o usuário especificar a transação
157 cout << "\nEnter charge (+) or payment (-): ";
158 double transaction; // cobrança ou pagamento
159 cin >> transaction;
160
161 // atualiza o saldo do registro
162 double oldBalance = client.getBalance();
163 client.setBalance(oldBalance + transaction);
164 outputLine(cout, client); // exibe o registro
165
166 // move ponteiro de posição de arquivo para corrigir o registro no arquivo
167 updateFile.seekp((accountNumber - 1) * sizeof(ClientData));
168
169 // grava o registro atualizado sobre o registro antigo no arquivo
170 updateFile.write(reinterpret_cast< const char * >(&client),
171 sizeof(ClientData));
172 } // fim do if
173 else // exibe erro se a conta não existir

```

Figura 17.15 Programa de conta bancária.

(continua)

```

174 cerr << "Account #" << accountNumber
175 << " has no information." << endl;
176 } // fim da função updateRecord
177
178 // cria e insere o registro
179 void newRecord(fstream &insertInFile)
180 {
181 // obtém o número da conta a ser criada
182 int accountNumber = getAccount("Enter new account number");
183
184 // move o ponteiro de posição de arquivo para corrigir o registro no arquivo
185 insertInFile.seekg((accountNumber - 1) * sizeof(ClientData));
186
187 // lê o registro no arquivo
188 ClientData client;
189 insertInFile.read(reinterpret_cast< char * >(&client),
190 sizeof(ClientData));
191
192 // cria o registro, se ele ainda não existir
193 if (client.getAccountNumber() == 0)
194 {
195 char lastName[15];
196 char firstName[10];
197 double balance;
198
199 // o usuário insere o sobrenome, o nome e o saldo
200 cout << "Enter lastname, firstname, balance\n? ";
201 cin >> setw(15) >> lastName;
202 cin >> setw(10) >> firstName;
203 cin >> balance;
204
205 // utiliza os valores para preencher os valores de conta
206 client.setLastName(lastName);
207 client.setFirstName(firstName);
208 client.setBalance(balance);
209 client.setAccountNumber(accountNumber);
210
211 // move o ponteiro de posição de arquivo para corrigir o registro no arquivo
212 insertInFile.seekp((accountNumber - 1) * sizeof(ClientData));
213
214 // insere o registro no arquivo
215 insertInFile.write(reinterpret_cast< const char * >(&client),
216 sizeof(ClientData));
217 } // fim do if
218 else // exibe erro se a conta já existir
219 {
220 cerr << "Account #" << accountNumber
221 << " already contains information." << endl;
222 } // fim da função newRecord
223
224 // exclui um registro existente
225 void deleteRecord(fstream &deleteFromFile)
226 {
227 // obtém o número de conta a excluir
228 int accountNumber = getAccount("Enter account to delete");
229
230 // move ponteiro de posição de arquivo para corrigir registro no arquivo

```

Figura 17.15 Programa de conta bancária.

(continua)

```

230 deleteFromFile.seekg((accountNumber - 1) * sizeof(ClientData));
231
232 // lê o registro no arquivo
233 ClientData client;
234 deleteFromFile.read(reinterpret_cast< char * >(&client),
235 sizeof(ClientData));
236
237 // exclui o registro, se existir no arquivo
238 if (client.getAccountNumber() != 0)
239 {
240 ClientData blankClient; // cria o registro em branco
241
242 // move o ponteiro de posição de arquivo para corrigir o registro no arquivo
243 deleteFromFile.seekp((accountNumber - 1) *
244 sizeof(ClientData));
245
246 // substitui registro existente pelo registro em branco
247 deleteFromFile.write(
248 reinterpret_cast< const char * >(&blankClient),
249 sizeof(ClientData));
250
251 cout << "Account #" << accountNumber << " deleted.\n";
252 } // fim do if
253 else // exibe erro se o registro não existir
254 cerr << "Account #" << accountNumber << " is empty.\n";
255 } // fim de deleteRecord
256
257 // exibe um único registro
258 void outputLine(ostream &output, const ClientData &record)
259 {
260 output << left << setw(10) << record.getAccountNumber()
261 << setw(16) << record.getLastName()
262 << setw(11) << record.getFirstName()
263 << setw(10) << setprecision(2) << right << fixed
264 << showpoint << record.getBalance() << endl;
265 } // fim da função outputLine
266
267 // obtém o valor do número da conta a partir do usuário
268 int getAccount(const char * const prompt)
269 {
270 int accountNumber;
271
272 // obtém valor do número da conta
273 do
274 {
275 cout << prompt << " (1 - 100): ";
276 cin >> accountNumber;
277 } while (accountNumber < 1 || accountNumber > 100);
278
279 return accountNumber;
280 } // fim da função getAccount

```

Figura 17.15 Programa de conta bancária.

(continuação)



O programa tem cinco opções (a opção 5 é para terminar o programa). A opção 1 chama a função `createTextFile` para armazenar uma lista formatada de todas as informações sobre a conta em um arquivo de texto chamado `print.txt` que pode ser impresso. A função `createTextFile` (linhas 100–135) aceita um objeto `fstream` como um argumento a fim de ser utilizado para inserir dados do arquivo `credit.dat`. A função `createTextFile` invoca a função-membro `istream read` (linhas 132–133) e utiliza as técnicas de acesso de arquivo sequencial da Figura 17.14 para inserir dados a partir de `credit.dat`. A função `outputLine`, discutida na Seção 17.10, é utilizada para gerar saída dos dados no arquivo `print.txt`. Observe que `createTextFile` utiliza a função-membro `istream seekg` (linha 117) para assegurar que o ponteiro de posição de arquivo está no começo do arquivo. Depois de escolher opção 1, o arquivo `print.txt` contém

| Account | Last Name | First Name | Balance |
|---------|-----------|------------|---------|
| 29      | Brown     | Nancy      | -24.54  |
| 33      | Dunn      | Stacey     | 314.33  |
| 37      | Barker    | Doug       | 0.00    |
| 88      | Smith     | Dave       | 258.34  |
| 96      | Stone     | Sam        | 34.98   |

A opção 2 chama `updateRecord` (linhas 138–176) para atualizar uma conta. Essa função atualiza somente um registro existente, assim, a função determina primeiro se o registro especificado está vazio. As linhas 148–149 lêem os dados no objeto `client`, utilizando a função-membro `istream read`. Em seguida, a linha 152 compara o valor retornado por `getAccountNumber` da estrutura `client` com zero para determinar se o registro contém informações. Se esse valor for zero, as linhas 174–175 imprimem uma mensagem de erro que indica que o registro está vazio. Se o registro contiver informações, a linha 154 exibe o registro, utilizando a função `outputLine`, a linha 159 insere a quantidade de transações, e as linhas 162–171 calculam o novo saldo e regravam o registro no arquivo. Uma saída típica para a opção 2 é

```
Enter account to update (1 - 100): 37
37 Barker Doug 0.00

Enter charge (+) or payment (-): +87.99
37 Barker Doug 87.99
```

A opção 3 chama a função `newRecord` (linhas 179–221) para adicionar uma nova conta ao arquivo. Se o usuário inserir um número de conta a uma conta existente, `newRecord` exibe uma mensagem de erro que indica que a conta já existe (linhas 219–220). Essa função adiciona uma nova conta da mesma maneira que o programa da Figura 17.12. Uma saída típica para a opção 3 é

```
Enter new account number (1 - 100): 22
Enter lastname, firstname, balance
? Johnston Sarah 247.45
```

A opção 4 chama a função `deleteRecord` (linhas 224–255) para excluir um registro do arquivo. A linha 227 solicita que o usuário insira o número de conta. Apenas um registro existente pode ser excluído, portanto, se a conta especificada estiver vazia, a linha 254 exibe uma mensagem de erro. Se a conta existir, as linhas 247–249 reinicializam essa conta copiando um registro vazio (`blankClient`) no arquivo. A linha 251 exibe uma mensagem para informar o usuário que o registro foi excluído. Uma saída típica para a opção 4 é

```
Enter account to delete (1 - 100): 29
Account #29 deleted.
```

Observe que a linha 43 abre o arquivo `credit.dat` criando um objeto `fstream` tanto para leitura como gravação, usando os modos `ios::in` e `ios::out` 'or-ed' juntos.

## 17.12 Entrada/saída de objetos

Este capítulo e o Capítulo 15 introduziram o estilo orientado a objetos de entrada/saída do C++. Entretanto, nossos exemplos concentraram-se em E/S de tipos de dados tradicionais em vez de objetos de tipos definidos pelo usuário. No Capítulo 11, mostramos como inserir e gerar saída de objetos utilizando sobrecarga de operadores. Realizamos a entrada de objeto sobrecarregando o operador de extração de fluxo `>>` para o `istream` apropriado. Realizamos a saída de objeto sobrecarregando o operador de inserção de fluxo `<<` para o `ostream` apropriado. Em ambos os casos, houve entrada e saída dos membros de dados de um objeto somente e, em cada caso, eles estavam em um formato significativo apenas para objetos desse tipo de dados abstrato particular. As funções-membro de um objeto não tiveram entrada e saída com os dados do objeto; em vez disso, uma cópia das funções-membro da classe permanece disponível internamente e é compartilhada por todos os objetos da classe.

Quando os membros de dados do objeto são enviados como saída para um arquivo de disco, perdemos as informações sobre o tipo do objeto. Armazenamos no disco apenas bytes de dados, não as informações de tipo. Se o programa que lê esses dados conhecer o tipo de objeto a que os dados correspondem, o programa lerá os dados em objetos desse tipo.

Um problema interessante ocorre quando armazenamos objetos de tipos diferentes no mesmo arquivo. Como podemos distingui-los (ou suas coleções de membros de dados) quando os lemos para um programa? O problema é que os objetos, em geral, não têm campos de tipo (estudamos essa questão cuidadosamente no Capítulo 13).

Uma abordagem seria fazer cada operador de saída sobrecarregado gerar saída de um código de tipo precedendo cada coleção de membros de dados que representa um objeto. Então a entrada de objeto sempre iniciaria lendo o campo de código de tipo e utilizando uma instrução `switch` para invocar a função sobrecarregada adequada. Embora essa solução não tenha a elegância da programação polimórfica, ela fornece um mecanismo praticável para reter objetos em arquivos e recuperá-los quando necessário.

## 17.13 Síntese

Neste capítulo, apresentamos várias técnicas de processamento de arquivo para manipular dados persistentes. Você aprendeu que os dados são armazenados em computadores na forma de 0s e 1s e que combinações desses valores formam bytes, campos, registros e, por fim, arquivos. Você foi apresentado às diferenças entre fluxos baseados em caractere e fluxos baseados em byte, e aos vários templates de classe de processamento de arquivo no arquivo de cabeçalho `<fstream>`. Então, você aprendeu a utilizar o processamento de arquivo seqüencial para manipular registros armazenados em ordem, pelo campo de chave de registro. Você também aprendeu a utilizar arquivos de acesso aleatório para recuperar e manipular instantaneamente registros de largura fixa. Por fim, apresentamos um estudo de caso de processamento de transação substancial que utiliza um arquivo de acesso aleatório para alcançar o processamento de acesso ‘instantâneo’. No próximo capítulo, discutiremos as típicas operações de manipulação de strings fornecidas pelo template da classe `basic_string`. Também introduziremos as capacidades de processamento de fluxo de string que permitem que a entrada e a saída de strings sejam realizadas a partir da e para a memória.

### Resumo

- Os arquivos são utilizados para persistência de dados — retenção permanente de grandes quantidades de dados.
- O menor item de dados que os computadores suportam é chamado de bit (abreviação de ‘*binary digit*’ — um dígito que pode assumir um de dois valores, 0 ou 1).
- Dígitos, letras e símbolos especiais são referidos como caracteres.
- O conjunto de todos os caracteres utilizados para escrever programas e representar itens de dados em um computador particular é chamado de conjunto de caracteres desse computador.
- Bytes são compostos de oito bits.
- Assim como os caracteres são compostos de bits, os campos são compostos de caracteres. Um campo é um grupo de caracteres que transmite algum significado.
- Em geral, um registro (isto é, uma classe em C++) é composto de vários campos (isto é, membros de dados em C++).
- Pelo menos um campo em um registro é escolhido como chave de registro para identificar um registro como pertencente a uma pessoa ou entidade particular que é distinto de todos os outros registros no arquivo.
- Em um arquivo seqüencial, os registros são normalmente armazenados ordenados por um campo de chave de registro.
- Um grupo de arquivos relacionados é freqüentemente armazenado em um banco de dados.
- Uma coleção de programas projetados para criar e gerenciar bancos de dados é chamada sistema de gerenciamento de bancos de dados (*database management system* – DBMS).
- O C++ visualiza cada arquivo como um fluxo seqüencial de bytes.
- Cada arquivo termina com um marcador de fim do arquivo ou em um número específico de byte registrado em uma estrutura de dados administrativa mantida pelo sistema.
- O cabeçalho `<fstream>` inclui as definições para os templates de classe de fluxo `basic_ifstream` (para entrada de arquivo), `basic_ofstream` (para saída de arquivo) e `basic_fstream` (para entrada e saída de arquivo).
- Para um objeto `ofstream`, o modo `file-open` pode ser `ios::out` para gerar saída de dados em um arquivo ou `ios::app` para acrescentar dados ao fim de um arquivo (sem modificar quaisquer dados já presentes no arquivo).
- O modo `file-open` `ios::ate` abre um arquivo para saída e move-se para o final do arquivo. Isso é normalmente utilizado para acrescentar dados a um arquivo, mas os dados podem ser gravados em qualquer lugar do arquivo.
- Os arquivos existentes abertos com o modo `ios::out` são truncados (isto é, todos os dados no arquivo são descartados).
- Por padrão, os objetos `ofstream` são abertos para saída.

- A função-membro `ofstream::open` abre um arquivo e o anexa a um objeto `ofstream` existente.
- Uma função-membro de operador `ios` sobrecarregado — `operator void*` — converte o fluxo em um ponteiro, assim, ele pode ser testado como 0 (isto é, o ponteiro nulo) ou não-zero (isto é, qualquer outro valor de ponteiro).
- Você pode utilizar a função-membro `ofstream::close` para fechar o objeto `ofstream` explicitamente.
- Tanto `istream` como `ostream` fornecem funções-membro para reposicionar o ponteiro de posição de arquivo (o número de bytes do próximo byte no arquivo a ser lido ou gravado). Essas funções-membro são `seekg` (*‘seek get’*) para `istream` e `seekp` (*‘seek put’*) para `ostream`.
- A direção de busca pode ser `ios::beg` (o padrão) para posicionamento em relação ao começo de um fluxo, `ios::cur` para posicionamento em relação à posição atual em um fluxo ou `ios::end` para posicionamento em relação ao fim de um fluxo.
- As funções-membro `tellg` e `tellp` são fornecidas para retornar as localizações atuais dos ponteiros *‘get’* e *‘put’*, respectivamente.
- Os registros individuais de um arquivo de acesso aleatório podem ser acessados diretamente (e de modo rápido) sem a necessidade de pesquisar outros registros.
- A função-membro `ostream::write` gera saída de um número fixo de bytes, começando em uma localização específica da memória, para o fluxo especificado. Quando o fluxo é associado com um arquivo, a função `write` grava os dados na localização do arquivo especificada pelo ponteiro de posição de arquivo *‘put’*.
- A função-membro `istream::read` transfere um número fixo de bytes a partir do fluxo especificado para uma área do início da memória em um endereço especificado. Se o fluxo estiver associado com um arquivo, a função `read` insere os bytes na localização do arquivo especificada pelo ponteiro de posição de arquivo *‘get’*.
- A função-membro `string::data` converte uma string em um array no estilo C terminado por caracteres não-nulos.

## Terminologia

|                                                      |                                                           |                                                                                             |
|------------------------------------------------------|-----------------------------------------------------------|---------------------------------------------------------------------------------------------|
| <fstream>, arquivo de cabeçalho                      | dígito binário                                            | nome de arquivo                                                                             |
| abrir um arquivo                                     | dígito decimal                                            | <code>open</code> , função-membro de <code>ofstream</code>                                  |
| aplicativo de acesso instantâneo                     | direção de busca                                          | persistência de dados                                                                       |
| arquivo                                              | dispositivo de armazenamento secundário                   | ponteiro de posição de arquivo                                                              |
| arquivo de acesso aleatório                          | fim do arquivo                                            | registro                                                                                    |
| arquivo sequencial                                   | <code>fstream</code>                                      | <code>seekg</code> <code>istream</code> , função-membro                                     |
| banco de dados                                       | função-membro <code>close</code> de <code>ofstream</code> | <code>seekp</code> <code>ostream</code> , função-membro                                     |
| bit                                                  | hierarquia de dados                                       | símbolo especial                                                                            |
| byte                                                 | <code>ios::app</code> , modo file-open                    | sistema de gerenciamento de bancos de dados<br>( <i>database management system – DBMS</i> ) |
| campo                                                | <code>ios::ate</code> , modo file-open                    | sistema de processamento de transação                                                       |
| campo de caractere                                   | <code>ios::beg</code> , ponto inicial de busca            | <code>size</code> , função de <code>string</code>                                           |
| chave de registro                                    | <code>ios::binary</code> , modo file-open                 | <code>tellg</code> <code>istream</code> , função-membro                                     |
| <code>cin</code> (entrada-padrão)                    | <code>ios::cur</code> , direção de busca                  | <code>tellp</code> <code>ostream</code> , função-membro                                     |
| <code>clog</code> (erro-padrão armazenado em buffer) | <code>ios::end</code> , direção de busca                  | truncar um arquivo existente                                                                |
| conjunto de caracteres                               | <code>ios::in</code> , modo file-open                     |                                                                                             |
| data, função de <code>string</code>                  | <code>ios::out</code> , modo file-open                    |                                                                                             |
| deslocamento a partir do início de um arquivo        | <code>ios::trunc</code> , modo file-open                  |                                                                                             |

## Exercícios de revisão

- 17.1** Preencha as lacunas em cada uma das seguintes sentenças:
- Em última instância, todos os itens de dados processados por um computador são reduzidos a combinações de \_\_\_\_\_ e \_\_\_\_\_.
  - O menor item de dados que um computador pode processar é chamado de \_\_\_\_\_.
  - Um \_\_\_\_\_ é um grupo de registros relacionados.
  - Dígitos, letras e símbolos especiais são referidos como \_\_\_\_\_.
  - Um grupo de arquivos relacionados é chamado de \_\_\_\_\_.
  - A função-membro \_\_\_\_\_ dos fluxos de arquivo `fstream`, `ifstream` e `ofstream` fecha um arquivo.
  - A função-membro `istream` \_\_\_\_\_ lê um caractere do fluxo especificado.
  - A função-membro \_\_\_\_\_ dos fluxos de arquivo `fstream`, `ifstream` e `ofstream` abre um arquivo.
  - A função-membro `istream` \_\_\_\_\_ é normalmente utilizada ao ler dados de um arquivo em aplicativos de acesso aleatório.
  - As funções-membro \_\_\_\_\_ e \_\_\_\_\_ de `istream` e `ostream` configuram o ponteiro de posição de arquivo como uma localização específica em uma entrada ou saída de fluxo, respectivamente.
- 17.2** Determine quais das seguintes sentenças são *verdadeiras* e quais são *falsas*. Se *falsa*, explique por quê.
- A função-membro `read` não pode ser utilizada para ler os dados do objeto de entrada `cin`.

- b) O programador deve criar os objetos `cin`, `cout`, `cerr` e `clog` explicitamente.
- c) Um programa deve chamar a função `close` explicitamente para fechar um arquivo associado com um objeto `ifstream`, `ofstream` ou `fstream`.
- d) Se o ponteiro de posição no arquivo aponta para uma localização em um arquivo seqüencial que não o início do arquivo, o arquivo deve ser fechado e reaberto para leitura a partir do início do arquivo.
- e) A função-membro `ostream::write` pode gravar no fluxo de saída-padrão `cout`.
- f) Os dados em arquivos seqüenciais sempre são atualizados sem sobrescrever dados próximos.
- g) É desnecessário pesquisar todos os registros em um arquivo de acesso aleatório para localizar um registro específico.
- h) Os registros em arquivos de acesso aleatório devem ter comprimento uniforme.
- i) As funções-membro `seekp` e `seekg` devem buscar em relação ao começo de um arquivo.

**17.3** Suponha que cada uma das seguintes instruções se aplica ao mesmo programa.

- a) Escreva uma instrução que abre o arquivo `oldmast.dat` para entrada; utilize um objeto `ifstream` chamado `inOldMaster`.
- b) Escreva uma instrução que abre o arquivo `trans.dat` para entrada; utilize um objeto `ifstream` chamado `inTransaction`.
- c) Escreva uma instrução que abre o arquivo `newmast.dat` para a saída (e criação); utilize o objeto `ofstream` `outNewMaster`.
- d) Escreva uma instrução que lê um registro do arquivo `oldmast.dat`. O registro consiste no inteiro `accountNumber`, na string `name` e no número de ponto flutuante `currentBalance`; utilize o objeto `ifstream` `inOldMaster`.
- e) Escreva uma instrução que lê um registro do arquivo `trans.dat`. O registro consiste no inteiro `accountNum` e no número de ponto flutuante `dollarAmount`; utilize o objeto `ifstream` `inTransaction`.
- f) Escreva uma instrução que grava um registro no arquivo `newmast.dat`. O registro consiste no inteiro `accountNum`, na string `name` e no número de ponto flutuante `currentBalance`; utilize o objeto `ofstream` `outNewMaster`.

**17.4** Localize o(s) erro(s) e mostre como corrija-lo(s) em cada uma das seguintes sentenças.

- a) O arquivo `payables.dat` referido pelo objeto `ofstream outPayable` não foi aberto.  

```
outPayable << account << company << amount << endl;
```
- b) A seguinte instrução deve ler um registro do arquivo `payables.dat`. O objeto `ifstream inPayable` referencia esse arquivo, e o objeto `istream inReceivable` referencia o arquivo `receivables.dat`.  

```
inReceivable >> account >> company >> amount;
```
- c) O arquivo `tools.dat` deve ser aberto para adicionar dados ao arquivo sem descartar os dados atuais.  

```
ofstream outTools("tools.dat", ios::out);
```

## Respostas dos exercícios de revisão

- 17.1** a) 1s, 0s. b) bit. c) arquivo. d) caracteres. e) banco de dados. f) `close`. g) `get`. h) `open`. i) `read`. j) `seekg`, `seekp`.
- 17.2**
  - a) Falsa. A função `read` pode ler de qualquer objeto de fluxo de entrada derivado de `istream`.
  - b) Falsa. Esses quatro fluxos são criados automaticamente para o programador. O cabeçalho `<iostream>` deve ser incluído em um arquivo para utilizá-los. Esse cabeçalho inclui declarações para cada objeto de fluxo.
  - c) Falsa. Os arquivos serão fechados quando os destrutores para os objetos `ifstream`, `ofstream` ou `fstream` executarem depois que os objetos de fluxo saírem do escopo, ou antes de a execução do programa terminar, mas é uma boa prática de programação fechar todos os arquivos explicitamente com `close` uma vez que eles não são mais necessários.
  - d) Falsa. A função-membro `seekp` ou `seekg` pode ser utilizada para reposicionar o ponteiro de posição de arquivo `put` ou `get` no começo do arquivo.
  - e) Verdadeira.
  - f) Falsa. Na maioria dos casos, os registros de arquivo seqüenciais não têm comprimento uniforme. Portanto, é possível que atualizar um registro faça com que outros dados sejam sobrescritos.
  - g) Verdadeira.
  - h) Falsa. Os registros em um arquivo de acesso aleatório são normalmente de comprimento uniforme.
  - i) Falsa. É possível buscar a partir do início do arquivo, do fim do arquivo e da posição atual no arquivo.
- 17.3**
  - a) `ifstream inOldMaster( "oldmast.dat", ios::in );`
  - b) `ifstream inTransaction( "trans.dat", ios::in );`
  - c) `ofstream outNewMaster( "newmast.dat", ios::out );`
  - d) `inOldMaster >> accountNumber >> name >> currentBalance;`
  - e) `inTransaction >> accountNum >> dollarAmount;`
  - f) `outNewMaster << accountNum << name << currentBalance;`
- 17.4**
  - a) Erro: O arquivo `payables.dat` não foi aberto antes de ser feita uma tentativa de enviar a saída dos dados para o fluxo.  
Correção: Utilize a função `ostream::open` para abrir `payables.dat` para a saída.
  - b) Erro: O objeto `istream` incorreto está sendo utilizado para ler um registro a partir do nome de arquivo `payables.dat`.  
Correção: Utilize o objeto `istream inPayable` para referenciar `payables.dat`.

- c) Erro: O conteúdo do arquivo é descartado porque o arquivo foi aberto para saída (`ios::out`).  
 Correção: Para adicionar dados ao arquivo, abra o arquivo para atualizar (`ios::ate`) ou para acrescentar (`ios::app`).

## Exercícios

**17.5** Preencha as lacunas em cada uma das seguintes sentenças:

- Os computadores armazenam grandes quantidades de dados em dispositivos de armazenamento secundários como \_\_\_\_\_.
- O \_\_\_\_\_ é composto de vários campos.
- Para facilitar a recuperação de registros específicos de um arquivo, um campo em cada registro é escolhido como \_\_\_\_\_.
- A vasta maioria das informações armazenadas em sistemas de computador é armazenada em arquivos \_\_\_\_\_.
- Um grupo de caracteres relacionados que transmite significado é chamado de \_\_\_\_\_.
- Os objetos de fluxo-padrão declarados pelo cabeçalho `<iostream>` são \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_ e \_\_\_\_\_.
- A função-membro `ostream` \_\_\_\_\_ gera saída de um caractere para o fluxo especificado.
- A função-membro `ostream` \_\_\_\_\_ é geralmente utilizada para gravar dados em um arquivo de acesso aleatório.
- A função-membro `istream` \_\_\_\_\_ reposiciona o ponteiro de posição de arquivo em um arquivo.

**17.6** Determine quais das seguintes sentenças são *verdadeiras* e quais são *falsas*. Se *falsa*, explique por quê.

- As impressionantes funções realizadas pelos computadores envolvem essencialmente a manipulação de zeros e uns.
- As pessoas preferem manipular bits em vez de caracteres e campos porque os bits são mais compactos.
- As pessoas especificam programas e itens de dados como caracteres; os computadores então manipulam e processam esses caracteres como grupos de zeros e uns.
- O CEP de 5 algarismos de uma pessoa é um exemplo de um campo numérico.
- O endereço de uma pessoa geralmente é considerado um campo alfabético em aplicativos de computador.
- Os itens de dados representados em computadores assumem uma hierarquia de dados em que itens de dados tornam-se cada vez maiores e mais complexos à medida que progredimos de campos para caracteres, então para bits etc.
- Uma chave de registro identifica um registro como pertencente a um campo particular.
- A maioria das empresas armazena todas as informações em um único arquivo para facilitar o processamento.
- Quando um programa cria um arquivo, o arquivo é automaticamente retido pelo computador para referência futura; isto é, diz-se que os arquivos são persistentes.

**17.7** O Exercício 17.3 pediu para o leitor escrever uma série de instruções únicas. De fato, essas instruções formam o núcleo de um importante tipo de programa processador de arquivo, a saber, um programa de correspondência de arquivo (*file-matching program*). Em processamento de dados comerciais, é comum ter vários arquivos em cada sistema de aplicativo. Em um sistema de contas a receber, por exemplo, em geral há um arquivo-mestre contendo informações detalhadas sobre cada cliente, como seu nome, endereço, número de telefone, saldo, limite de crédito, termos de desconto, arranjos de contrato e possivelmente um histórico condensado de compras recentes e pagamentos de conta.

Quando as transações ocorrem (por exemplo, as vendas e os pagamentos à vista são feitos), elas são inseridas em um arquivo. No final de cada período de negócio (um mês para algumas empresas, uma semana para outras e um dia em alguns casos), o arquivo de transações (chamado `trans.dat` no Exercício 17.3) é aplicado ao arquivo-mestre (chamado `oldmast.dat` no Exercício 17.3), atualizando assim o registro de compras e pagamentos de cada conta. Durante uma operação de atualização, o arquivo-mestre é regravado como um novo arquivo (`newmast.dat`), que é então utilizado no final do próximo período de negócios para iniciar o processo de atualização novamente.

Programas de correspondência de arquivo devem lidar com certos problemas que não existem em programas de um único arquivo. Por exemplo, nem sempre ocorre uma correspondência. Um cliente no arquivo-mestre poderia não ter feito nenhuma compra ou pagamentos em dinheiro no período atual de negócios e portanto nenhum registro para esse cliente aparecerá no arquivo de transação. De maneira semelhante, um cliente que fez alguma compra ou pagamentos em dinheiro poderia ter se mudado de endereço e a empresa poderia não ter tido uma oportunidade de criar um registro-mestre para esse cliente.

Utilize as instruções no Exercício 17.3 como uma base para escrever um programa de correspondência de arquivo completo de contas a receber. Utilize o número de conta em cada arquivo como a chave de registro para propósitos de correspondência. Suponha que cada arquivo é um arquivo seqüencial com registros armazenados em ordem crescente de número de conta.

Quando ocorre uma correspondência (isto é, registros com o mesmo número de conta aparecem em arquivos-mestre e arquivos de transação), adicione a quantia em dólar no arquivo de transação ao saldo atual do arquivo-mestre e grave o registro `newmast.dat`. (Suponha que as compras sejam indicadas por quantias positivas no arquivo de transação e os pagamentos, por quantias negativas.) Quando houver um registro-mestre para uma conta particular, mas nenhum registro de transação correspondente, simplesmente grave o registro-mestre em `newmast.dat`. Quando houver um registro de transação, mas não o registro-mestre correspondente, imprima a mensagem "Unmatched transaction record for account number ..." ["Não há transação correspondente ao número de conta ..."] (preencha o número de conta a partir do registro de transação).

- 17.8** Depois de escrever o programa do Exercício 17.7, escreva um programa simples para criar alguns dados de teste para conferir o programa. Utilize os seguintes dados de conta de exemplo:

| Número de conta | Nome       | Saldo  |
|-----------------|------------|--------|
| 100             | Alan Jones | 348,17 |
| 300             | Mary Smith | 27,19  |
| 500             | Sam Sharp  | 0,00   |
| 700             | Suzy Green | -14,22 |

| Número de conta do arquivo de transação | Quantia da transação |
|-----------------------------------------|----------------------|
| 100                                     | 27,14                |
| 300                                     | 62,11                |
| 400                                     | 100,56               |
| 900                                     | 82,17                |

- 17.9** Execute o programa do Exercício 17.7, utilizando os arquivos de dados de teste criados no Exercício 17.8. Imprima o novo arquivo-mestre. Verifique se as contas foram atualizadas corretamente.
- 17.10** É possível (de fato, é comum) ter vários registros de transação com a mesma chave de registro. Isso ocorre porque um cliente particular talvez faça várias compras e pagamentos em dinheiro durante um período de negócios. Reescreva seu programa de correspondência de arquivo de contas a receber do Exercício 17.7 para oferecer a possibilidade de tratamento de vários registros de transação com a mesma chave de registro. Modifique os dados de teste do Exercício 17.8 para incluir os seguintes registros adicionais de transação:

| Número de conta | Quantia em dólar |
|-----------------|------------------|
| 300             | 83,89            |
| 700             | 80,78            |
| 700             | 1,53             |

- 17.11** Escreva uma série de instruções que realizam cada uma das seguintes tarefas. Suponha que definimos a classe `Person` que contém membros de dados `private`

```
char lastName[15];
char firstName[15];
char age[4];
```

e as funções-membro `public`

```
// funções de acesso para sobrenome
void setLastName(string);
string getLastName() const;

// funções de acesso para primeiro nome
void setFirstName(string);
string getFirstName() const;

// funções de acesso para idade
void setAge(string);
string getAge() const;
```

Também suponha que todo arquivo de acesso aleatório tenha sido aberto adequadamente.

- Inicialize o arquivo `nameage.dat` com 100 registros que armazenam os valores `lastName = "unassigned"`, `firstName = ""` e `age = "0"`.
- Insira 10 sobrenomes, nomes e idades e grave-os no arquivo.
- Atualize um registro que já contém informações. Se o registro não contiver informações, informe ao usuário "No info".
- Exclua um registro que contém informações reiniciando esse registro particular.



- 17.12** Você é o proprietário de uma loja de equipamentos de construção e precisa manter um inventário que pode lhe informar as diferentes ferramentas que você tem, a quantidade de cada item à disposição e o preço correspondente. Escreva um programa que inicializa o arquivo de acesso aleatório `hardware.dat` para 100 registros vazios, permita que você próprio insira os dados relativos a cada ferramenta, permita listar todas as suas ferramentas, permita a exclusão de um registro de uma ferramenta que você não tem mais e permita que você atualize *qualquer* informação no arquivo. O número de identificação de ferramenta deve ser o número de registro. Utilize as seguintes informações para iniciar seu arquivo:

| Nº do registro | Nome de ferramenta | Quantidade | Preço |
|----------------|--------------------|------------|-------|
| 3              | Lixadeira          | 7          | 57,98 |
| 17             | Martelo            | 76         | 11,99 |
| 24             | Serra tico-tico    | 21         | 11,00 |
| 39             | Cortador de grama  | 3          | 79,50 |
| 56             | Serra elétrica     | 18         | 99,99 |
| 68             | Chave de fenda     | 106        | 6,99  |
| 77             | Marreta            | 11         | 21,50 |
| 83             | Chave inglesa      | 34         | 7,50  |

- 17.13** (*Gerador de palavras para números de telefone*) Teclados de telefone padrão contêm os dígitos de 0 a 9. Os números de 2 a 9 têm, cada um, três letras associadas, como é indicado pela seguinte tabela:

| Dígito | Letra |
|--------|-------|
| 2      | A B C |
| 3      | D E F |
| 4      | G H I |
| 5      | J K L |
| 6      | M N O |
| 7      | P R S |
| 8      | T U V |
| 9      | W X Y |

Muitas pessoas acham difícil memorizar números de telefone, então utilizam a correspondência entre dígitos e letras para criar palavras de sete letras que correspondem aos seus números de telefone. Por exemplo, uma pessoa cujo número de telefone é 686-2377 poderia utilizar a correspondência indicada na tabela acima para forjar a palavra de sete letras “NUMBERS”.

Com frequência, as empresas tentam obter números de telefone fáceis de memorizar. Se uma empresa puder anunciar um número simples de discar para seus clientes, sem dúvida receberá um número maior de ligações.

Cada palavra de sete letras corresponde a exatamente um número de telefone de sete dígitos. O restaurante que deseja aumentar seu negócio de entregas em domicílio (*‘takeout’*, em inglês) seguramente poderia fazer isso com o número 825-3688 (isto é, *‘TAKEOUT’*).

Cada número de telefone de sete letras corresponde a muitas palavras diferentes de sete letras. Infelizmente, a maioria representa justaposições irreconhecíveis de letras. É possível, porém, que o proprietário de um salão de cabeleireiro ficasse satisfeito em saber que o número de telefone de seu salão, 424-7288, corresponde a *‘HAIRCUT’* (corte de cabelo, em inglês). O proprietário de uma loja de bebidas, sem dúvida, ficaria encantado em descobrir que o número de telefone da loja, 233-7226, corresponde a *‘BEERCAN’* (lata de cerveja, em inglês). Um veterinário com o número de telefone 738-2273 gostaria de saber que seu número corresponde à palavra de sete letras *‘PETCARE’* (cuidado de animais de estimação).

Escreva um programa que, dado um número de sete dígitos, grava em um arquivo cada possível palavra de sete letras correspondentes a esse número. Há 2187 (3 à sétima potência) dessas palavras. Evite números de telefone com os dígitos 0 e 1.

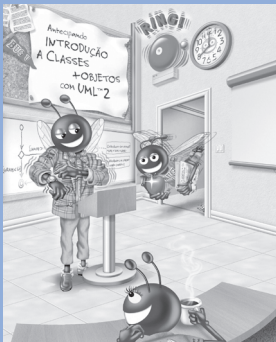
- 17.14** Escreva um programa que utiliza o operador `sizeof` para determinar os tamanhos em bytes dos vários tipos de dados no seu sistema de computador. Grave os resultados no arquivo `datasize.dat`, de modo que você possa imprimir os resultados posteriormente. Os



resultados devem ser exibidos no formato de duas colunas com o nome do tipo na coluna esquerda e o tamanho do tipo na coluna direita, como em:

|                    |    |
|--------------------|----|
| char               | 1  |
| unsigned char      | 1  |
| short int          | 2  |
| unsigned short int | 2  |
| int                | 4  |
| unsigned int       | 4  |
| long int           | 4  |
| unsigned long int  | 4  |
| float              | 4  |
| double             | 8  |
| long double        | 10 |

[Nota: Os tamanhos dos tipos de dados predefinidos no seu computador podem diferir dos listados.]



*A diferença entre a palavra quase certa e a certa é realmente um assunto bastante extenso — é a mesma diferença que há entre o vaga-lume (lightning bug) e o relâmpago (lightning).*

Mark Twain

*Escrevi uma carta mais longa que o normal, porque me falta tempo para fazê-la mais curta.*

Blaise Pascal

*Não diga nada sobre o segredo que você sabe.*

Miguel de Cervantes

*Ajuste a ação à palavra, a palavra à ação; tendo sempre em mira não ultrapassar a modéstia da natureza.*

William Shakespeare

## Classe string e processamento de fluxo de string

### OBJETIVOS

Neste capítulo, você aprenderá:

- Como utilizar a classe `string` da C++ Standard Library para tratar strings como objetos completos com todos os recursos.
- A atribuir, concatenar, comparar, pesquisar e trocar strings.
- Como determinar as características de string.
- A localizar, substituir e inserir caracteres em uma string.
- A converter strings em strings no estilo C e vice-versa.
- Como utilizar iteradores string.
- A realizar entrada a partir de e saída para strings na memória.

- 18.1** Introdução
- 18.2** Atribuição e concatenação de strings
- 18.3** Comparando strings
- 18.4** Substrings
- 18.5** Trocando strings
- 18.6** Características de string
- 18.7** Localizando strings e caracteres em uma string
- 18.8** Substituindo caracteres em uma string
- 18.9** Inserindo caracteres em uma string
- 18.10** Conversão para strings `char *` baseadas em ponteiro no estilo C
- 18.11** Iteradores
- 18.12** Processamento de fluxo de string
- 18.13** Síntese

Resumo | Terminologia | Exercícios de revisão | Respostas dos exercícios de revisão | Exercícios

## 18.1 Introdução

O template de classe `basic_string` do C++ fornece operações de manipulação de strings típicas como copiar, pesquisar etc. A definição de template e todos os recursos de suporte são definidos no namespace `std`; esses incluem a instrução `typedef`

```
typedef basic_string< char > string;
```

que cria o tipo de alias `string` para `basic_string< char >`. Um `typedef` também é oferecido para o tipo `wchar_t`<sup>1</sup>. O tipo `wchar_t` armazena caracteres (por exemplo, caracteres de dois bytes, caracteres de quatro bytes etc.) para suportar outros conjuntos de caracteres. Utilizamos `string` exclusivamente por todo este capítulo. Para utilizar strings, inclua o arquivo de cabeçalho `<string>`.

Um objeto `string` pode ser inicializado com um argumento de construtor como

```
string text("Hello"); // cria string a partir de const char *
```

que cria uma `string` contendo o caractere em "Hello" ou com dois argumentos de construtor como em

```
string name(8, 'x'); // string de 8 caracteres 'x'
```

que cria uma `string` contendo oito caracteres 'x'. A classe `string` também fornece um construtor-padrão (que cria uma `string` vazia) e um construtor de cópia. Uma `string` vazia é uma `string` que não contém caracteres.

Uma `string` também pode ser inicializada via a sintaxe de construção alternativa na definição de uma `string` como em

```
string month = "March"; // mesmo que: string month("March");
```

Lembre-se de que o operador `=` na declaração anterior não é uma atribuição; em vez disso, é uma chamada implícita ao construtor da classe `string`, que faz a conversão.

Observe que a classe `string` não fornece conversões de `int` ou `char` para `string` em uma definição `string`. Por exemplo, as definições

```
string error1 = 'c';
string error2('u');
string error3 = 22;
string error4(8);
```

resultam em erro de sintaxe. Observe que é permitido atribuir um único caractere a um objeto `string` em uma instrução de atribuição como em

```
string1 = 'n';
```



### Erro comum de programação 18.1

*Tentar converter um `int` ou `char` em uma `string` por meio de uma inicialização em uma declaração ou via um argumento de construtor é um erro de compilação.*

<sup>1</sup> O tipo `wchar_t` é comumente utilizado para representar caracteres Unicode®, que são de 16 bits, mas o tamanho de `wchar_t` não é fixado pelo padrão. O Unicode Standard oferece uma especificação para produzir uma codificação consistente dos caracteres e símbolos mais comuns de todo o mundo. Para aprender mais sobre o padrão Unicode, visite [www.unicode.org](http://www.unicode.org).

Diferentemente das strings `char *` no estilo C, as strings não são necessariamente terminadas por caractere nulo. [Nota: O documento C++ padrão fornece somente uma descrição da interface para a classe `string` — a implementação é dependente de plataforma.] O comprimento de uma `string` pode ser recuperado com a função-membro `length` e com a função-membro `size`. O operador de subscrito, `[]`, pode ser utilizado com strings para acessar e modificar caracteres individuais. Como strings no estilo C, strings têm um primeiro subscrito de 0 e um último subscrito de `length() - 1`.

A maioria das funções-membro `string` aceita como argumentos uma localização de subscrito inicial e o número de caracteres sobre os quais operar.

O operador de extração de fluxo (`>>`) é sobrecarregado para suportar strings. A instrução

```
string stringObject;
cin >> stringObject;
```

lê uma string a partir do dispositivo de entrada-padrão. A entrada é delimitada por caracteres de espaço em branco. Quando um delimitador é encontrado, a operação de entrada é terminada. A função `getline` também é sobrecarregada para suportar strings. A instrução

```
string string1;
getline(cin, string1);
```

lê uma string a partir do teclado em `string1`. A entrada é delimitada por um caractere de nova linha (`'\n'`), assim `getline` pode ler uma linha de texto e a transferir para um objeto `string`.

## 18.2 Atribuição e concatenação de strings

A Figura 18.1 demonstra a atribuição e a concatenação de strings. A linha 7 inclui o cabeçalho `<string>` para a classe `string`. As strings `string1`, `string2` e `string3` são criadas nas linhas 12–14. A linha 16 atribui o valor de `string1` a `string2`. Depois da atribuição, `string2` é uma cópia de `string1`. A linha 17 utiliza a função-membro `assign` para copiar `string1` para `string3`. Uma cópia separada é feita (isto é, `string1` e `string3` são objetos independentes). A classe `string` também fornece uma versão sobrecarregada da função-membro `assign` que copia um número especificado de caracteres, como em

```
targetString.assign(stringOriginal, início, númeroDeCaracteres);
```

onde `stringOriginal` é a string a ser copiada, `início` é o subscrito inicial e `númeroDeCaracteres` é o número de caracteres a copiar.

A linha 22 utiliza o operador de subscrito para atribuir `'r'` a `string3[ 2 ]` (formando "car") e atribuir `'r'` a `string2[ 0 ]` (formando "rat"). A saída de strings é então gerada.

As linhas 28–29 geram a saída do conteúdo de `string3` um caractere por vez utilizando a função-membro `at`. A função-membro `at` fornece **acesso verificado** (ou **verificação de intervalos**); isto é, ultrapassar o final de `string` lança uma exceção `out_of_range`. (Consulte o Capítulo 16 para uma discussão detalhada sobre o tratamento de exceções.) Observe que o operador de subscrito, `[]`, não fornece acesso verificado. Isso é consistente com sua utilização em arrays.

```
1 // Figura 18.1: Fig18_01.cpp
2 // Demonstrando a atribuição e a concatenação de strings.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string>
8 using std::string;
9
10 int main()
11 {
12 string string1("cat");
13 string string2;
14 string string3;
15
16 string2 = string1; // atribui string1 a string2
17 string3.assign(string1); // atribui string1 a string3
18 cout << "string1: " << string1 << "\nstring2: " << string2
19 << "\nstring3: " << string3 << "\n\n";
20 }
```

**Figura 18.1** Demonstrando a atribuição e a concatenação de strings.

(continua)

```

21 // modifica string2 e string3
22 string2[0] = string3[2] = 'r';
23
24 cout << "After modification of string2 and string3:\n" << "string1: "
25 << string1 << "\nstring2: " << string2 << "\nstring3: ";
26
27 // demonstrando a função-membro em
28 for (int i = 0; i < string3.length(); i++)
29 cout << string3.at(i);
30
31 // declara string4 e string5
32 string string4(string1 + "apult"); // concatenação
33 string string5;
34
35 // += sobrecarregado
36 string3 += "pet"; // cria "carpet"
37 string1.append("acomb"); // cria "catacomb"
38
39 // acrescenta localizações de subscrito 4 até o final de string1 para
40 // criar a string "comb" (a string5 estava inicialmente vazia)
41 string5.append(string1, 4, string1.length() - 4);
42
43 cout << "\n\nAfter concatenation:\nstring1: " << string1
44 << "\nstring2: " << string2 << "\nstring3: " << string3
45 << "\nstring4: " << string4 << "\nstring5: " << string5 << endl;
46 return 0;
47 } // fim de main

```

```

string1: cat
string2: cat
string3: cat

```

After modification of string2 and string3:

```

string1: cat
string2: rat
string3: car

```

After concatenation:

```

string1: catacomb
string2: rat
string3: carpet
string4: catapult
string5: comb

```

**Figura 18.1** Demonstrando a atribuição e a concatenação de strings.

(continuação)



## Erro comum de programação 18.2

Acessar um subscrito de *string* fora dos limites da função *string* utilizando *at* é um erro de lógica que causa uma exceção *out\_of\_range*.



## Erro comum de programação 18.3

Acessar um elemento além do tamanho da *string* utilizando o operador de subscrito é um erro de lógica não informado.

A string *string4* é declarada (linha 32) e inicializada com o resultado da concatenação de *string1* e "apult" utilizando o operador de adição sobrecarregado, +, o que para a classe *string* denota concatenação. A linha 36 utiliza o operador de atribuição de adição, +=, para concatenar *string3* e "pet". A linha 37 utiliza a função-membro **append** para concatenar *string1* e "acomb".

A linha 41 acrescenta a string "comb" à string `string5` vazia. Essa função-membro é passada à string (`string1`) a partir da qual recupera caracteres, o subscrito inicial na string (4) e o número de caracteres a acrescentar (o valor retornado por `string1.length() - 4`).

### 18.3 Comparando strings

A classe `string` fornece funções-membro para comparar strings. A Figura 18.2 demonstra as capacidades de comparação da classe `string`.

O programa declara quatro strings com as linhas 12–15 e gera a saída de cada string (linhas 17–18). A condição na linha 21 testa a igualdade de `string1` contra `string4` utilizando o operador de igualdade sobrecarregado. Se a condição for `true`, a saída de "`string1 == string4`" é gerada. Se a condição é `false`, a condição na linha 25 é testada. Todas as funções de operador sobrecarregado da classe `string` demonstradas aqui, bem como as não demonstradas (`!=`, `<`, `>=` e `<=`), retornam valores `bool`.

```

1 // Figura 18.2: Fig18_02.cpp
2 // Demonstrando capacidades de comparação de string.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string>
8 using std::string;
9
10 int main()
11 {
12 string string1("Testing the comparison functions.");
13 string string2("Hello");
14 string string3("stinger");
15 string string4(string2);
16
17 cout << "string1: " << string1 << "\nstring2: " << string2
18 << "\nstring3: " << string3 << "\nstring4: " << string4 << "\n\n";
19
20 // comparando string1 e string4
21 if (string1 == string4)
22 cout << "string1 == string4\n";
23 else // string1 != string4
24 {
25 if (string1 > string4)
26 cout << "string1 > string4\n";
27 else // string1 < string4
28 cout << "string1 < string4\n";
29 } // fim de else
30
31 // comparando string1 e string2
32 int result = string1.compare(string2);
33
34 if (result == 0)
35 cout << "string1.compare(string2) == 0\n";
36 else // resultado != 0
37 {
38 if (result > 0)
39 cout << "string1.compare(string2) > 0\n";
40 else // resultado < 0
41 cout << "string1.compare(string2) < 0\n";
42 } // fim de else

```

Figura 18.2 Comparando strings.

(continua)

```

43
44 // comparando string1 (elementos 2-5) e string3 (elementos 0-5)
45 result = string1.compare(2, 5, string3, 0, 5);
46
47 if (result == 0)
48 cout << "string1.compare(2, 5, string3, 0, 5) == 0\n";
49 else // resultado != 0
50 {
51 if (result > 0)
52 cout << "string1.compare(2, 5, string3, 0, 5) > 0\n";
53 else // resultado < 0
54 cout << "string1.compare(2, 5, string3, 0, 5) < 0\n";
55 } // fim de else
56
57 // comparando string2 e string4
58 result = string4.compare(0, string2.length(), string2);
59
60 if (result == 0)
61 cout << "string4.compare(0, string2.length(), "
62 << "string2) == 0" << endl;
63 else // resultado != 0
64 {
65 if (result > 0)
66 cout << "string4.compare(0, string2.length(), "
67 << "string2) > 0" << endl;
68 else // resultado < 0
69 cout << "string4.compare(0, string2.length(), "
70 << "string2) < 0" << endl;
71 } // fim de else
72
73 // comparando string2 e string4
74 result = string2.compare(0, 3, string4);
75
76 if (result == 0)
77 cout << "string2.compare(0, 3, string4) == 0" << endl;
78 else // resultado != 0
79 {
80 if (result > 0)
81 cout << "string2.compare(0, 3, string4) > 0" << endl;
82 else // resultado < 0
83 cout << "string2.compare(0, 3, string4) < 0" << endl;
84 } // fim de else
85
86 return 0;
87 } // fim de main

```

```

string1: Testing the comparison functions.
string2: Hello
string3: stinger
string4: Hello

```

```

string1 > string4
string1.compare(string2) > 0
string1.compare(2, 5, string3, 0, 5) == 0
string4.compare(0, string2.length(), string2) == 0
string2.compare(0, 3, string4) < 0

```

**Figura 18.2** Comparando strings.

(continuação)



A linha 32 utiliza a função-membro `string::compare` para comparar `string1` com `string2`. À variável `result` é atribuído 0 se as strings forem equivalentes, um número positivo se `string1` for **lexicograficamente** maior que `string2` ou um número negativo se `string1` for lexicograficamente menor que `string2`. Como uma string que inicia com 'T' é considerada lexicograficamente maior que uma string que inicia com 'H', `result` recebe um valor maior que 0, como confirmado pela saída. Um léxico é um dicionário. Quando dizemos que uma string é lexicograficamente menor do que outra, queremos dizer que a primeira string é alfabeticamente menor que a segunda. O computador utiliza o mesmo critério que você utilizaria para colocar em ordem alfabética uma lista de nomes.

A linha 45 utiliza uma versão sobrecarregada da função-membro `compare` para comparar partes de `string1` e `string3`. Os dois primeiros argumentos (2 e 5) especificam o subscrito inicial e o comprimento da parte de `string1` ("sting") a comparar com `string3`. O terceiro argumento é a string de comparação. Os dois últimos argumentos (0 e 5) são o subscrito inicial e o comprimento da parte da string de comparação sendo comparada (também "sting"). O valor atribuído a `result` é 0 quanto à igualdade, um número positivo se `string1` for lexicograficamente maior que `string3` ou negativo se `string1` for lexicograficamente menor que `string3`. Como as duas partes de strings sendo comparadas aqui são idênticas, o valor atribuído a `result` é 0.

A linha 58 utiliza outra versão sobrecarregada da função `compare` para comparar `string4` e `string2`. Os dois primeiros argumentos são os mesmos — o subscrito inicial e o comprimento. O último argumento é a string para comparação. O valor retornado também é o mesmo — 0 para a igualdade, um número positivo se `string4` for lexicograficamente maior que `string2` ou um número negativo se `string4` for lexicograficamente menor que `string2`. Como as duas partes de strings sendo comparadas aqui são idênticas, o valor atribuído a `result` é 0.

A linha 74 chama a função-membro `compare` para comparar os 3 primeiros caracteres em `string2` com `string4`. Como "Hel" é menor que "Hello", um valor menor que zero é retornado.

## 18.4 Substrings

A classe `string` fornece a função-membro `substr` para recuperar uma substring de uma string. O resultado é um novo objeto `string` que é copiado a partir da string de origem. A Figura 18.3 demonstra `substr`.

O programa declara e inicializa uma string na linha 12. A linha 16 utiliza a função-membro `substr` para recuperar uma substring de `string1`. O primeiro argumento especifica o subscrito inicial da substring desejada; o segundo argumento especifica o comprimento da substring.

## 18.5 Trocando strings

A classe `string` fornece a função-membro `swap` para trocar strings. A Figura 18.4 troca duas strings.

As linhas 12–13 declaram e inicializam as strings `first` e `second`. A saída de cada string é, então, gerada. A linha 18 utiliza a função-membro `string::swap` para trocar os valores de `first` e `second`. As duas strings são impressas novamente para confirmar que, de fato, elas foram trocadas. A função-membro `string::swap` é útil para implementar programas que classificam strings.

```

1 // Figura 18.3: Fig18_03.cpp
2 // Demonstrando a função-membro string::substr.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string>
8 using std::string;
9
10 int main()
11 {
12 string string1("The airplane landed on time.");
13
14 // recupera a substring "plane" que
15 // inicia no subscrito 7 e consiste em 5 elementos
16 cout << string1.substr(7, 5) << endl;
17 return 0;
18 } // fim de main

```

plane

**Figura 18.3** Demonstrando a função-membro `string::substr`.

```

1 // Figura 18.4: Fig18_04.cpp
2 // Utilizando a função swap para fazer a troca de duas strings.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string>
8 using std::string;
9
10 int main()
11 {
12 string first("one");
13 string second("two");
14
15 // gera a saída das strings
16 cout << "Before swap:\n first: " << first << "\nsecond: " << second;
17
18 first.swap(second); // troca strings
19
20 cout << "\n\nAfter swap:\n first: " << first
21 << "\nsecond: " << second << endl;
22 return 0;
23 } // fim de main

```

Before swap:  
first: one  
second: two

After swap:  
first: two  
second: one

**Figura 18.4** Utilizando a função swap para trocar duas strings.

## 18.6 Características de string

A classe string fornece funções-membro para reunir informações sobre o tamanho, o comprimento, a capacidade, o comprimento máximo e outras características de uma string. O tamanho ou comprimento de uma string é o número de caracteres atualmente armazenados na string. A **capacidade** de uma string é o número de caracteres que podem ser armazenados na string sem alocar mais memória. A capacidade de uma string deve ser pelo menos igual ao tamanho atual da string, embora possa ser maior. A capacidade exata de uma string depende da implementação. O **tamanho máximo** é o maior tamanho possível que uma string pode ter. Se esse valor for excedido, uma exceção `length_error` será lançada. A Figura 18.5 demonstra funções-membro da classe string para determinar várias características de strings.

O programa declara a string `string1` vazia (linha 16) e a passa para a função `printStatistics` (linha 19). A função `printStatistics` (linhas 49–55) aceita uma referência a uma `const string` como um argumento e gera a saída da capacidade (utilizando a função-membro `capacity`), do tamanho máximo (utilizando a função-membro `max_size`), do tamanho (utilizando a função-membro `size`), do comprimento (utilizando a função-membro `length`) e se a string está vazia (utilizando a função-membro `empty`). A chamada inicial a `printStatistics` indica que os valores iniciais para a capacidade, o tamanho e o comprimento de `string1` são 0.

O tamanho e o comprimento de 0 indicam que não há caracteres armazenados em string. Como a capacidade inicial é 0, quando caracteres forem colocados em `string1`, a memória será alocada para acomodar os novos caracteres. Lembre-se de que o tamanho e o comprimento são sempre idênticos. Nessa implementação, o tamanho máximo é 4294967293. O objeto `string1` é uma string vazia, desse modo, a função `empty` retorna `true`.

A linha 23 lê uma string a partir da linha de comando. Nesse exemplo, "tomato soup" é inserido. Como um caractere de espaço em branco é um delimitador, somente "tomato" é armazenado em `string1`; entretanto, "soup" permanece no buffer de entrada. A linha 27 chama a função `printStatistics` para gerar saída das estatísticas de `string1`. Note na saída que o comprimento é 6 e que a capacidade é 15.

```

1 // Figura 18.5: Fig18_05.cpp
2 // Demonstrando funções-membro relacionadas ao tamanho e à capacidade.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::cin;
7 using std::boolalpha;
8
9 #include <string>
10 using std::string;
11
12 void printStatistics(const string &);
13
14 int main()
15 {
16 string string1;
17
18 cout << "Statistics before input:\n" << boolalpha;
19 printStatistics(string1);
20
21 // lê somente "tomato" de "tomato soup"
22 cout << "\n\nEnter a string: ";
23 cin >> string1; // delimitado por espaço em branco
24 cout << "The string entered was: " << string1;
25
26 cout << "\nStatistics after input:\n";
27 printStatistics(string1);
28
29 // lê "soup"
30 cin >> string1; // delimitado por espaço em branco
31 cout << "\n\nThe remaining string is: " << string1 << endl;
32 printStatistics(string1);
33
34 // acrescenta 46 caracteres a string1
35 string1 += "1234567890abcdefghijklmnopqrstuvwxyz1234567890";
36 cout << "\n\nstring1 is now: " << string1 << endl;
37 printStatistics(string1);
38
39 // adiciona 10 elementos a string1
40 string1.resize(string1.length() + 10);
41 cout << "\n\nStats after resizing by (length + 10):\n";
42 printStatistics(string1);
43
44 cout << endl;
45 return 0;
46 } // fim de main
47
48 // exibe a estatística de string
49 void printStatistics(const string &stringRef)
50 {
51 cout << "capacity: " << stringRef.capacity() << "\nmax size: "
52 << stringRef.max_size() << "\nsize: " << stringRef.size()
53 << "\nlength: " << stringRef.length()
54 << "\nempty: " << stringRef.empty();
55 } // fim de printStatistics

```

**Figura 18.5** Imprimindo características de string.

(continua)

```

Statistics before input:
capacity: 0
max size: 4294967293
size: 0
length: 0
empty: true

Enter a string: tomato soup
The string entered was: tomato
Statistics after input:
capacity: 15
max size: 4294967293
size: 6
length: 6
empty: false

The remaining string is: soup
capacity: 15
max size: 4294967293
size: 4
length: 4
empty: false

string1 is now: soup1234567890abcdefghijklmnopqrstuvwxyz1234567890
capacity: 63
max size: 4294967293
size: 50
length: 50
empty: false

Stats after resizing by (length + 10):
capacity: 63
max size: 4294967293
size: 60
length: 60
empty: false

```

**Figura 18.5** Imprimindo características de string.

(continuação)



### Dica de desempenho 18.1

*Para minimizar o número de vezes que a memória é alocada e desalocada, algumas implementações da classe `string` fornecem uma capacidade-padrão muito além do comprimento da `string`.*

A linha 30 lê "soup" a partir do buffer de entrada e o armazena em `string1`, substituindo, assim, "tomato". A linha 32 passa `string1` para `printStats`.

A linha 35 utiliza o operador `+=` sobrecarregado para concatenar uma string com 46 caracteres de comprimento para `string1`. A linha 37 passa `string1` para `printStats`. Note que a capacidade aumentou para 63 elementos e o comprimento é agora 50.

A linha 40 utiliza a função-membro `resize` para aumentar o comprimento de `string1` em 10 caracteres. Os elementos adicionais são configurados como caracteres nulos. Note que, na saída, a capacidade não se alterou e o comprimento é agora 60.

## 18.7 Localizando strings e caracteres em uma string

A classe `string` fornece as funções-membro `const` para localizar substrings e caracteres em uma `string`. A Figura 18.6 demonstra as funções `find`.

A string `string1` é declarada e inicializada na linha 12. A linha 17 tenta localizar "is" em `string1` utilizando a função `find`. Se "is" for localizado, o subscrito da localização inicial dessa string é retornado. Se a string não for localizada, o valor `string::npos` (uma constante `public static` definida na classe `string`) é retornado. Esse valor é retornado pelas funções relacionadas com `string` `find` para indicar que uma substring ou um caractere não foi localizado na string.

A linha 18 utiliza a função-membro `rfind` para pesquisar `string1` de trás para a frente (isto é, da direita para a esquerda). Se "is" for localizada, a localização de subscrito é retornada. Se a string não for localizada, `string::npos` é retornada. [Nota: As demais funções `find` apresentadas nesta seção retornam o mesmo tipo, a menos que notado diferentemente.]

A linha 21 utiliza a função-membro `find_first_of` para localizar a primeira ocorrência em `string1` de qualquer caractere em "misop". A pesquisa é feita desde o início de `string1`. O caractere 'o' é localizado no elemento 1.

A linha 26 utiliza a função-membro `find_last_of` para localizar a última ocorrência em `string1` de qualquer caractere em "misop". A pesquisa é feita a partir do fim de `string1`. O caractere 'o' é localizado no elemento 29.

A linha 31 utiliza a função-membro `find_first_not_of` para localizar o primeiro caractere em `string1` não contido em "noi spm". O caractere 'l' é localizado no elemento 8. A pesquisa é feita desde o início de `string1`.

A linha 37 utiliza a função-membro `find_first_not_of` para localizar o primeiro caractere não contido em "12noi spm". O caractere '.' é localizado no elemento 12. A pesquisa é feita a partir do fim de `string1`.

As linhas 43–44 utilizam a função-membro `find_first_not_of` para localizar o primeiro caractere não contido em "noon is 12 pm; midnight is not.". Nesse caso, a string sendo pesquisada contém cada caractere especificado no argumento de string. Como um caractere não foi localizado, `string::npos` (que tem o valor -1 nesse caso) é retornada.

```

1 // Figura 18.6: Fig18_06.cpp
2 // Demonstrando as funções-membro string find.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string>
8 using std::string;
9
10 int main()
11 {
12 string string1("noon is 12 pm; midnight is not.");
13 int location;
14
15 // localiza "is" nas posições 5 e 25
16 cout << "Original string:\n" << string1
17 << "\n\n(find) \"is\" was found at: " << string1.find("is")
18 << "\n\n(rfind) \"is\" was found at: " << string1.rfind("is");
19
20 // localiza 'o' na posição 1
21 location = string1.find_first_of("misop");
22 cout << "\n\n(find_first_of) found '" << string1[location]
23 << "' from the group \"misop\" at: " << location;
24
25 // localiza 'o' na posição 29
26 location = string1.find_last_of("misop");
27 cout << "\n\n(find_last_of) found '" << string1[location]
28 << "' from the group \"misop\" at: " << location;
29
30 // localiza 'l' na posição 8
31 location = string1.find_first_not_of("noi spm");
32 cout << "\n\n(find_first_not_of) '" << string1[location]
33 << "' is not contained in \"noi spm\" and was found at:"
34 << location;
35
36 // localiza '.' na posição 12
37 location = string1.find_first_not_of("12noi spm");
38 cout << "\n\n(find_first_not_of) '" << string1[location]
39 << "' is not contained in \"12noi spm\" and was "
40 << "found at:" << location << endl;

```

Figura 18.6 Demonstrando as funções string find.

(continua)

```

41
42 // procura caracteres não em string1
43 location = string1.find_first_not_of(
44 "noon is 12 pm; midnight is not.");
45 cout << "\nfind_first_not_of(\"noon is 12 pm; midnight is not.\")"
46 << " returned: " << location << endl;
47 return 0;
48 } // fim de main

```

Original string:

noon is 12 pm; midnight is not.

(find) "is" was found at: 5

(rfind) "is" was found at: 25

(find\_first\_of) found 'o' from the group "misop" at: 1

(find\_last\_of) found 'o' from the group "misop" at: 29

(find\_first\_not\_of) 'l' is not contained in "noi spm" and was found at:8

(find\_first\_not\_of) '.' is not contained in "12noi spm" and was found at:12

find\_first\_not\_of("noon is 12 pm; midnight is not.") returned: -1

**Figura 18.6** Demonstrando as funções string find.

(continuação)

## 18.8 Substituindo caracteres em uma string

A Figura 18.7 demonstra as funções-membro string para substituir e apagar caracteres. As linhas 13–17 declaram e inicializam a string `string1`. A linha 23 utiliza a função-membro string `erase` para apagar tudo a partir do (e incluindo o) caractere na posição 62 até o final de `string1`. [Nota: Cada caractere de nova linha ocupa um elemento na string.]

As linhas 29–36 utilizam `find` para localizar cada ocorrência do caractere de espaço em branco. Cada espaço então é substituído por um ponto com uma chamada à função-membro string `replace`. A função `replace` aceita três argumentos: o subscrito do caractere na string em que a substituição deve iniciar, o número de caracteres a substituir e a string substituta. A função-membro `find` retorna `string::npos` quando o caractere de pesquisa não é localizado. Na linha 35, 1 é adicionado a `position` para continuar pesquisando na localização do próximo caractere.

```

1 // Figura 18.7: Fig18_07.cpp
2 // Demonstrando as funções-membro string erase e replace.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string>
8 using std::string;
9
10 int main()
11 {
12 // o compilador concatena todas as partes em uma string
13 string string1("The values in any left subtree"
14 "\nare less than the value in the"
15 "\nparent node and the values in"

```

**Figura 18.7** Demonstrando as funções `erase` e `replace`.

(continua)

```

16 "\nany right subtree are greater"
17 "\nthan the value in the parent node");
18
19 cout << "Original string:\n" << string1 << endl << endl;
20
21 // remove todos os caracteres a partir da (e incluindo a)
22 // localização 62 até o final de string1
23 string1.erase(62);
24
25 // gera a saída da nova string
26 cout << "Original string after erase:\n" << string1
27 << "\n\nAfter first replacement:\n";
28
29 int position = string1.find(" "); // localiza o primeiro espaço
30
31 // substitui todos os espaços pelo ponto
32 while (position != string::npos)
33 {
34 string1.replace(position, 1, ".");
35 position = string1.find(" ", position + 1);
36 } // fim do while
37
38 cout << string1 << "\n\nAfter second replacement:\n";
39
40 position = string1.find("."); // localiza o primeiro ponto
41
42 // substitui todos os pontos por dois ponto-e-vírgulas
43 // NOTA: isso irá sobrescrever os caracteres
44 while (position != string::npos)
45 {
46 string1.replace(position, 2, "xxxxx;yyy", 5, 2);
47 position = string1.find(".", position + 1);
48 } // fim do while
49
50 cout << string1 << endl;
51 return 0;
52 } // fim de main

```

Original string:

The values in any left subtree  
are less than the value in the  
parent node and the values in  
any right subtree are greater  
than the value in the parent node

Original string after erase:

The values in any left subtree  
are less than the value in the

After first replacement:

The.values.in.any.left.subtree  
are.less.than.the.value.in.the

After second replacement:

The;;alues;;n;;ny;;eft;;ubtree  
are;;ess;;han;;he;;alue;;n;;he

**Figura 18.7** Demonstrando as funções erase e replace.

(continuação)



As linhas 40–48 utilizam a função `find` para localizar cada ponto e outra função sobrecarregada `replace` para substituir cada ponto e seu caractere seguinte por dois caracteres de ponto-e-vírgula. Os argumentos passados a essa versão de `replace` são o subscrito do elemento onde a operação substituição inicia, o número de caracteres a substituir, uma string de caractere substituta a partir da qual uma substring é selecionada para ser utilizada como caracteres substitutos, o elemento na string de caractere em que a substring substituta inicia e o número de caracteres na string de caracteres substituta a utilizar.

## 18.9 Inserindo caracteres em uma string

A classe `string` fornece as funções-membro para inserir caracteres em uma string. A Figura 18.8 demonstra as capacidades de `string` `insert`.

```

1 // Figura 18.8: Fig18_08.cpp
2 // Demonstrando as funções-membro insert da classe string.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string>
8 using std::string;
9
10 int main()
11 {
12 string string1("beginning end");
13 string string2("middle ");
14 string string3("12345678");
15 string string4("xx");
16
17 cout << "Initial strings:\nstring1: " << string1
18 << "\nstring2: " << string2 << "\nstring3: " << string3
19 << "\nstring4: " << string4 << "\n\n";
20
21 // insere "middle" na localização 10 de string1
22 string1.insert(10, string2);
23
24 // "xx" de inserção na localização 3 de string3
25 string3.insert(3, string4, 0, string::npos);
26
27 cout << "Strings after insert:\nstring1: " << string1
28 << "\nstring2: " << string2 << "\nstring3: " << string3
29 << "\nstring4: " << string4 << endl;
30 return 0;
31 } // fim de main

```

Initial strings:  
string1: beginning end  
string2: middle  
string3: 12345678  
string4: xx

Strings after insert:  
string1: beginning middle end  
string2: middle  
string3: 123xx45678  
string4: xx

**Figura 18.8** Demonstrando as funções-membro `string` `insert`.

O programa declara, inicializa e, então, gera a saída das strings `string1`, `string2`, `string3` e `string4`. A linha 22 utiliza a função-membro `string::insert` para inserir o conteúdo de `string2` antes do elemento 10 da `string1`.

A linha 25 utiliza `insert` para inserir `string4` antes do elemento 3 da `string3`. Os dois últimos argumentos especificam o elemento inicial e final de `string4` que devem ser inseridos. Utilizar `string::npos` faz com que a string inteira seja inserida.

## 18.10 Conversão para strings char \* baseadas em ponteiro no estilo C

A classe `string` fornece funções-membro para converter os objetos da classe `string` em strings baseadas em ponteiro no estilo C. Como mencionado anteriormente, ao contrário das strings baseadas em ponteiro, as strings não são necessariamente terminadas por caractere nulo. Essas funções de conversão são úteis quando uma dada função aceita uma string baseada em ponteiro como um argumento. A Figura 18.9 demonstra a conversão de strings em strings baseadas em ponteiro.

```

1 // Figura 18.9: Fig18_09.cpp
2 // Convertendo em strings no estilo C.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string>
8 using std::string;
9
10 int main()
11 {
12 string string1("STRINGS"); // construtor de string com char* arg
13 const char *ptr1 = 0; // inicializa *ptr1
14 int length = string1.length();
15 char *ptr2 = new char[length + 1]; // incluindo nulo
16
17 // copia caracteres de string1 na memória alocada
18 string1.copy(ptr2, length, 0); // copia string1 para ptr2 char*
19 ptr2[length] = '\0'; // adiciona terminador nulo
20
21 cout << "string string1 is " << string1
22 << "\nstring1 converted to a C-Style string is "
23 << string1.c_str() << "\nptr1 is ";
24
25 // Atribui ao ponteiro ptr1 a const char * retornada pela
26 // função data(). NOTA: essa é uma atribuição potencialmente
27 // perigosa. Se string1 for modificada, o ponteiro ptr1 pode
28 // tornar-se inválido.
29 ptr1 = string1.data();
30
31 // gera a saída de cada caractere utilizando ponteiro
32 for (int i = 0; i < length; i++)
33 cout << *(ptr1 + i); // utiliza aritmética de ponteiro
34
35 cout << "\nptr2 is " << ptr2 << endl;
36 delete [] ptr2; // reivindica memória dinamicamente alocada
37 return 0;
38 } // fim de main

```

```

string string1 is STRINGS
string1 converted to a C-Style string is STRINGS
ptr1 is STRINGS
ptr2 is STRINGS

```

**Figura 18.9** Convertendo strings em strings no estilo C e arrays de caracteres.

O programa declara uma string, um int e dois ponteiros char (linhas 12–15). A string `string1` é inicializada como "STRINGS", `ptr1` é inicializado como 0 e `length` é inicializado como o comprimento de `string1`. Memória de tamanho suficiente para armazenar um equivalente de string baseada em ponteiro `string` `string1` é alocada dinamicamente e anexada ao ponteiro char `ptr2`.

A linha 18 utiliza a função-membro string `copy` para copiar o objeto `string1` para o array char apontado por `ptr2`. A linha 19 coloca manualmente um caractere de terminação nulo no array apontado por `ptr2`.

A linha 23 utiliza a função `c_str` para copiar o objeto `string1` e adiciona, automaticamente, um caractere de terminação nulo. Essa função retorna um `const char *` cuja saída é gerada pelo operador de inserção de fluxo.

A linha 29 atribui ao `const char * ptr1` um ponteiro retornado pela função-membro `data` da classe `string`. Essa função-membro retorna um array no estilo C terminado por caractere não-nulo. Observe que não modificamos a string `string1` nesse exemplo. Se `string1` tivesse de ser modificada (por exemplo, a memória dinâmica de string muda seu endereço devido a uma chamada de função-membro como `string1.insert( 0, "abcd" );`), `ptr1` poderia tornar-se inválido — o que poderia produzir resultados imprevisíveis.

As linhas 32–33 utilizam a aritmética de ponteiro para gerar a saída do array de caracteres apontado por `ptr1`. Nas linhas 35–36, a saída da string no estilo C apontada por `ptr2` é gerada e a memória alocada para `ptr2` é excluída para evitar um vazamento de memória.



## Erro comum de programação 18.4

*Não terminar o array de caracteres retornado por `data` com um caractere nulo pode gerar erros de tempo de execução.*



## Boa prática de programação 18.1

*Sempre que possível, utilize os objetos da classe `string` mais robustos em vez de strings baseadas em ponteiro no estilo C.*

### 18.1.1 Iteradores

A classe `string` fornece iteradores para percorrer strings para a frente e para trás. Os iteradores fornecem acesso aos caracteres individuais com uma sintaxe semelhante às operações de ponteiro. Não há verificação de intervalos para iteradores. Observe que nesta seção fornecemos ‘exemplos mecânicos’ para demonstrar o uso de iteradores. Discutimos usos mais robustos de iteradores no Capítulo 23. A Figura 18.10 demonstra os iteradores.

As linhas 12–13 declaram a string `string1` e `string::const_iterator` `iterator1`. Um `const_iterator` é um iterador que não pode modificar a string — nesse caso a string — pela qual ele está iterando. O iterador `iterator1` é inicializado para o início de `string1` com a função-membro string da classe `begin`. Há duas versões de `begin` — uma que retorna um iterador para iterar por uma não-const string e uma versão `const` que retorna um `const_iterator` para iterar por uma const string. A linha 15 gera saída de `string1`.

As linhas 19–23 utilizam o iterador de `iterator1` para ‘percorrer’ `string1`. A função-membro `end` da classe `string` retorna um iterador (ou um `const_iterator`) para a posição além do último elemento de `string1`. Cada elemento é impresso desreferenciando o iterador de maneira muito parecida a como você desreferenciaria um ponteiro, e o iterador é avançado uma posição utilizando o operador `++`.

A classe `string` fornece as funções-membro `rend` e `rbegin` para acessar caracteres string individuais no sentido inverso a partir do fim de uma string para seu início. As funções-membro `rend` e `rbegin` podem retornar `reverse_iterators` e `const_reverse_iterators` (com base no fato de a string ser não-const ou const). Nos exercícios, solicitamos ao leitor escrever um programa que demonstra essas capacidades. Utilizaremos mais iteradores e iteradores invertidos no Capítulo 23.

```

1 // Figura 18.10: Fig18_10.cpp
2 // Utilizando um iterador para gerar a saída de uma string.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string>
8 using std::string;
9
10 int main()
11 {

```

**Figura 18.10** Utilizando um iterador para gerar a saída de uma string.

(continua)

```

12 string string1("Testing iterators");
13 string::const_iterator iterator1 = string1.begin();
14
15 cout << "string1 = " << string1
16 << "\n(Using iterator iterator1) string1 is: ";
17
18 // itera pela string
19 while (iterator1 != string1.end())
20 {
21 cout << *iterator1; // desreferencia o iterador para obter char
22 iterator1++; // avança o iterador para o próximo char
23 } // fim do while
24
25 cout << endl;
26 return 0;
27 } // fim de main

```

```

string1 = Testing iterators
(Using iterator iterator1) string1 is: Testing iterators

```

**Figura 18.10** Utilizando um iterador para gerar a saída de uma string.

(continuação)



### Dica de prevenção de erro 18.1

Utilize a função-membro `string::at` (em vez de iteradores) quando você quiser o benefício de verificação de intervalos.



### Boa prática de programação 18.2

Quando as operações que envolvem o iterador não devem modificar os dados sendo processados, utilize um `const_iterator`. Esse é outro exemplo de empregar o princípio do menor privilégio.

## 18.12 Processamento de fluxo de string

Além da E/S de fluxo padrão e da E/S de fluxo de arquivo, a E/S do fluxo C++ inclui capacidades para realizar entrada a partir de e saída para strings na memória. Essas capacidades são muitas vezes referidas como **E/S na memória** ou **processamento de fluxo de string**.

A entrada a partir de uma string é suportada pela classe `istream`. A saída para uma string é suportada pela classe `ostream`. Os nomes de classe `istream` e `ostream` são, na realidade, aliases definidos pelo `typedef`

```

typedef basic_istream< char > istream;
typedef basic_ostream< char > ostream;

```

Os templates da classe `basic_istream` e `basic_ostream` fornecem a mesma funcionalidade que as classes `istream` e `ostream` mais outras funções-membro específicas à formatação na memória. Os programas que utilizam formatação na memória devem incluir os arquivos de cabeçalho `<sstream>` e `<iostream>`.

Uma aplicação dessas técnicas é a validação de dados. Um programa pode ler uma linha inteira por vez a partir do fluxo de entrada em uma string. Em seguida, uma rotina de validação pode examinar cuidadosamente o conteúdo da string e corrigir (ou reparar) os dados, se necessário. Então o programa pode prosseguir para a entrada a partir da string, sabendo que os dados de entrada estão no formato adequado.

Gerar saída para uma string é uma maneira elegante de tirar proveito das poderosas capacidades de formatação de saída de fluxos C++. Os dados podem ser preparados em uma string para simular o formato de tela editado. Essa string poderia ser gravada em um arquivo de disco para preservar a imagem de tela.

Um objeto `ostream` utiliza um objeto string para armazenar os dados de saída. A função-membro `str` da classe `ostream` retorna uma cópia dessa string.

A Figura 18.11 demonstra um objeto `ostream`. O programa cria o objeto `ostream` `outputString` (linha 15) e utiliza o operador de inserção de fluxo para gerar a saída de uma série de strings e valores numéricos para o objeto.

As linhas 27–28 geram saída de `string string1`, `string string2`, `string string3`, `double double1`, `string string4`, `int integer`, `string string5` e do endereço de `int integer` — todos para `outputString` na memória. A linha 31 utiliza o operador de inserção de fluxo e a chamada `outputString.str()` para exibir uma cópia da string criada nas linhas 27–28. A linha 34 demonstra

que mais dados podem ser acrescentados à string na memória simplesmente emitindo outra operação de inserção de fluxo para `outputString`. As linhas 35–36 exibem a string `outputString` depois de acrescentar caracteres adicionais.

```

1 // Figura 18.11: Fig18_11.cpp
2 // Utilizando um objeto ostream dinamicamente alocado.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string>
8 using std::string;
9
10 #include <sstream> // arquivo de cabeçalho para processamento de fluxo de string
11 using std::ostringstream; // operadores de inserção de fluxo
12
13 int main()
14 {
15 ostream outputString; // cria instância ostream
16
17 string string1("Output of several data types ");
18 string string2("to an ostream object:");
19 string string3("\n double: ");
20 string string4("\n int: ");
21 string string5("\naddress of int: ");
22
23 double double1 = 123.4567;
24 int integer = 22;
25
26 // gera saída de strings, double e int para ostream outputString
27 outputString << string1 << string2 << string3 << double1
28 << string4 << integer << string5 << &integer;
29
30 // chama str para obter o conteúdo de string de ostream
31 cout << "outputString contains:\n" << outputString.str();
32
33 // adiciona mais caracteres e chama str para gerar a saída da string
34 outputString << "\nmore characters added";
35 cout << "\n\nafter additional stream insertions,\n"
36 << "outputString contains:\n" << outputString.str() << endl;
37 return 0;
38 } // fim de main

```

```

outputString contains:
Output of several data types to an ostream object:
 double: 123.457
 int: 22
address of int: 0012F540

after additional stream insertions,
outputString contains:
Output of several data types to an ostream object:
 double: 123.457
 int: 22
address of int: 0012F540
more characters added

```

**Figura 18.11** Utilizando um objeto `ostringstream` dinamicamente alocado.

Um objeto `istream` insere dados a partir de uma string na memória para variáveis do programa. Dados são armazenados em um objeto `istream` como caracteres. A entrada a partir do objeto `istream` funciona de modo idêntico à entrada a partir de qualquer arquivo. O fim da string é interpretado pelo objeto `istream` como fim do arquivo.

A Figura 18.12 demonstra a entrada de um objeto `istream`. As linhas 15–16 criam a string `input` contendo os dados e o objeto `istream` `inputString` construído para conter os dados em string `input`. A string `input` contém os dados

Input test 123 4.7 A"

os quais, quando lidos como entrada para o programa, consistem em duas strings ("Input" e "test"), um `int` (123), um `double` (4.7) e um `char` ('A'). Esses caracteres são extraídos para as variáveis `string1`, `string2`, `integer`, `double1` e `character` na linha 23.

A saída dos dados é então realizada nas linhas 25–28. O programa tenta ler a partir de `inputString` novamente na linha 32. A condição `if` na linha 35 utiliza a função `good` (Seção 15.8) para testar se restaram quaisquer dados. Como não restaram dados, a função retorna `false` e a parte `else` da instrução `if...else` é executada.

```

1 // Figura 18.12: Fig18_12.cpp
2 // Demonstrando a entrada a partir de um objeto istream.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <string>
8 using std::string;
9
10 #include <sstream>
11 using std::istringstream;
12
13 int main()
14 {
15 string input("Input test 123 4.7 A");
16 istringstream inputString(input);
17 string string1;
18 string string2;
19 int integer;
20 double double1;
21 char character;
22
23 inputString >> string1 >> string2 >> integer >> double1 >> character;
24
25 cout << "The following items were extracted\n"
26 << "from the istringstream object:" << "\nstring: " << string1
27 << "\nstring: " << string2 << "\n int: " << integer
28 << "\ndouble: " << double1 << "\n char: " << character;
29
30 // tenta ler a partir do fluxo vazio
31 long value;
32 inputString >> value;
33
34 // testa resultados do fluxo
35 if (inputString.good())
36 cout << "\n\nlong value is: " << value << endl;
37 else
38 cout << "\n\ninputString is empty" << endl;
39
40 return 0;
41 } // fim de main

```

**Figura 18.12** Demonstrando a entrada a partir de um objeto `istream`.

(continua)

```

The following items were extracted
from the istream object:
string: Input
string: test
 int: 123
double: 4.7
 char: A

inputString is empty

```

**Figura 18.12** Demonstrando a entrada a partir de um objeto `istream`.

(continuação)

### 18.13 Síntese

Este capítulo introduziu a classe `string` da C++ Standard Library, que permite aos programas tratar strings como objetos completos com todos os recursos. Discutimos atribuição, concatenação, comparação, pesquisa e troca de strings. Também introduzimos vários métodos para determinar características de string, para localizar, substituir e inserir caracteres em uma string e para converter strings em strings no estilo C e vice-versa. Você também aprendeu sobre iteradores de string e a entrada a partir de e a saída para strings na memória. No próximo capítulo, você aprenderá a escrever scripts de CGI com C++ para criar páginas Web dinâmicas com o C++.

#### Resumo

- O template de classe `basic_string` do C++ fornece operações de manipulação de string típicas como copiar, pesquisar etc.
- A instrução `typedef`

```
typedef basic_string< char > string;
```

 cria o tipo alias `string` para `basic_string< char >`. Um `typedef` também é oferecido para o tipo `wchar_t`. O tipo `wchar_t` normalmente armazena caracteres de dois bytes (16 bits) para suportar outros conjuntos de caracteres. O tamanho de `wchar_t` não é corrigido pelo padrão.
- Para utilizar strings, inclua o arquivo de cabeçalho `<string>` da C++ Standard Library.
- A classe `string` não fornece nenhum construtor que converte de `int` ou `char` para `string`.
- É permitido atribuir um único caractere a um objeto `string` em uma instrução de atribuição.
- strings não são necessariamente terminadas por caractere nulo.
- A maioria das funções-membro `string` aceita como argumentos uma localização de subscrito inicial e o número de caracteres sobre os quais operar.
- A classe `string` fornece o operador `=` sobrecarregado e a função-membro `assign` para atribuições de `string`.
- O operador de subscrito, `[]`, fornece acesso de leitura/gravação para qualquer elemento de uma `string`.
- A função-membro `string::at` fornece acesso verificado — ultrapassar qualquer extremidade da `string` lança uma exceção `out_of_range`. O operador de subscrito, `[]`, não fornece acesso verificado.
- A classe `string` fornece os operadores `+` e `+=` sobrecarregados e a função-membro `append` para realizar a concatenação `string`.
- A classe `string` fornece os operadores `==`, `!=`, `<`, `>`, `<=` e `>=` sobrecarregados para comparações de `string`.
- A função-membro `string::compare` compara duas strings (ou substrings) e retorna 0 se as strings forem iguais, um número positivo se a primeira string for lexicograficamente maior que a segunda ou um número negativo se a primeira string for lexicograficamente menor que a segunda.
- A função-membro `string::substr` recupera uma substring a partir de uma `string`.
- A função-membro `string::swap` troca o conteúdo de duas strings.
- As funções-membro `string::size` e `length` retornam o tamanho ou o comprimento de uma `string` (isto é, o número de caracteres atualmente armazenados na `string`).
- A função-membro `string::capacity` retorna o número total de caracteres que podem ser armazenados na `string` sem aumentar a quantidade de memória alocada para a `string`.
- A função-membro `string::max_size` retorna o tamanho máximo que uma `string` pode ter.
- A função-membro `string::resize` altera o comprimento de uma `string`.



- As funções `find` da classe `string` `find`, `rfind`, `find_first_of`, `find_last_of` e `find_first_not_of` localizam substrings ou caracteres em uma `string`.
- A função-membro `string::erase` exclui elementos de uma `string`.
- A função-membro `string::replace` substitui caracteres em uma `string`.
- A função-membro `string::insert` insere caracteres em uma `string`.
- A função-membro `string::c_str` retorna um `const char *` que aponta para uma `string` de caracteres no estilo C terminada por caractere nulo que contém todos os caracteres em uma `string`.
- A função-membro `string::data` retorna um `const char *` que aponta para um array de caracteres no estilo C terminado por caractere não-nulo que contém todos os caracteres em uma `string`.
- A classe `string` fornece as funções-membro `end` e `begin` para iterar por elementos individuais.
- A classe `string` fornece as funções-membro `rend` e `rbegin` para acessar caracteres `string` individuais no sentido inverso, a partir do fim de uma `string` para o começo.
- A entrada a partir de uma `string` é suportada pelo tipo `istream`. A saída para uma `string` é suportada pelo tipo `ostream`.
- A função-membro `ostream::str` retorna uma cópia `string` de uma `string`.

## Terminologia

<sstream>, arquivo de cabeçalho

acesso verificado

`append`, função-membro da classe `string`

`assign`, função-membro da classe `string`

`at`, função-membro da classe `string`

`basic_string`, template de classe

`begin`, função-membro da classe `string`

`c_str`, função-membro da classe `string`

capacidade de uma `string`

`capacity`, função-membro da classe `string`

comparação lexicográfica

`compare`, função-membro da classe `string`

comprimento de uma `string`

`const_iterator`

`const_reverse_iterator`

`copy`, função-membro da classe `string`

`data`, função-membro da classe `string`

E/S na memória

`end`, função-membro da classe `string`

`erase`, função-membro da classe `string`

`find`, função-membro da classe `string`

`find_first_not_of`, função-membro da classe `string`

`find_first_of`, função-membro da classe `string`

`find_last_of`, função-membro da classe `string`

`getline`, função-membro da classe `string`

`insert`, função-membro da classe `string`

`istream`, classe

`iterator`

`length`, função-membro da classe `string`

`max_size`, função-membro da classe `string`

`ostream`, classe

processamento de fluxo `string`

`rbegin`, função-membro da classe `string`

`rend`, função-membro da classe `string`

`replace`, função-membro da classe `string`

`resize`, função-membro da classe `string`

`reverse_iterator`

`rfind`, função-membro da classe `string`

`size`, função-membro da classe `string`

`str`, função-membro da classe `ostream`

`string` vazia

`string::npos`, constante

`substr`, função-membro da classe `string`

`swap`, função-membro da classe `string`

tamanho máximo de uma `string`

verificação de intervalo

`wchar_t`, tipo

## Exercícios de revisão

**18.1** Preencha as lacunas em cada uma das seguintes sentenças:

- O cabeçalho \_\_\_\_\_ deve ser incluído para a classe `string`.
- A classe `string` pertence ao namespace \_\_\_\_\_.
- A função \_\_\_\_\_ exclui caracteres de uma `string`.
- A função \_\_\_\_\_ localiza a primeira ocorrência de qualquer caractere de uma `string`.

**18.2** Determine quais das seguintes sentenças são *verdadeiras* e quais são *falsas*. Se uma instrução for *falsa*, explique por quê.

- A concatenação de objetos `string` pode ser realizada com o operador de atribuição de adição, `+=`.
- Os caracteres dentro de uma `string` iniciam no índice 0.
- O operador de atribuição, `=`, copia uma `string`.
- Uma `string` no estilo C é um objeto `string`.

**18.3** Localize o(s) erro(s) em cada uma das seguintes instruções e explique como corrija-lo(s):

- a) `string string1(28); // constrói string1`  
`string string2('z'); // constrói string2`
- b) // supõe o namespace std é conhecido  
`const char *ptr = name.data(); // nome é "joe bob"`  
`ptr[ 3 ] = '-';`  
`cout << ptr << endl;`

## Respostas dos exercícios de revisão

**18.1** a) `<string>`. b) `std`. c) `erase`. d) `find_first_of`.

**18.2** a) Verdadeira.  
 b) Verdadeira.  
 c) Verdadeira.  
 d) Falsa. Uma `string` é um objeto que fornece muitos serviços diferentes. Uma `string` no estilo C não fornece nenhum serviço. As `strings` no estilo C são terminadas por caractere nulo; `strings` não são necessariamente terminadas por caractere nulo. As `strings` no estilo C são ponteiros; mas `strings` não o são.

**18.3** a) Não existem construtores da classe `string` para argumentos do tipo inteiro e argumentos de caractere. Outros construtores válidos devem ser utilizados — converter os argumentos para `strings` se houver necessidade.  
 b) A função `data` não adiciona um terminador null. Além disso, o código tenta modificar um `const char`. Substitua todas as linhas pelo código:

```
cout << name.substr(0, 3) + "-" + name.substr(4) << endl;
```

## Exercícios

**18.4** Preencha as lacunas em cada uma das seguintes sentenças:

- a) As funções-membro \_\_\_\_\_ e \_\_\_\_\_ da classe `string` convertem `strings` em `strings` no estilo C.
- b) A função-membro \_\_\_\_\_ da classe `string` é utilizada para atribuição.
- c) \_\_\_\_\_ é o tipo de retorno da função `rbegin`.
- d) A função-membro \_\_\_\_\_ da classe `string` é utilizada para recuperar uma substring.

**18.5** Determine quais das seguintes sentenças são *verdadeiras* e quais são *falsas*. Se for *falsa*, explique por quê.

- a) As `strings` são sempre terminadas por caractere nulo.
- b) A função-membro `max_size` da classe `string` retorna o tamanho máximo de uma `string`.
- c) A função-membro `at` da classe `string` pode lançar uma exceção `out_of_range`.
- d) A função-membro `begin` da classe `string` retorna um `iterator`.

**18.6** Localize quaisquer erros a seguir e explique como corrija-los:

- a) `std::cout << s.data() << std::endl; // s é "hello"`
- b) `erase( s.rfind( "x" ), 1 ); // s é "xenon"`
- c) `string& foo()`  

```
{
 string s("Hello");
 ... // outras instruções
 return;
} // fim da função foo
```

**18.7** (*Criptografia simples*) Algumas informações sobre a Internet podem ser criptografadas com um algoritmo simples conhecido como 'rot13', que troca cada caractere por um correspondente 13 posições à frente no alfabeto. Portanto, 'a' corresponde a 'n', e 'x' corresponde a 'k'. rot13 é um exemplo de **criptografia de chave simétrica**. Com a criptografia de chave simétrica, tanto o encriptador como o decriptador utilizam a mesma chave.

- a) Escreva um programa que criptografa uma mensagem utilizando rot13.
- b) Escreva um programa que decripta a mensagem embaralhada utilizando 13 como a chave.
- c) Depois de escrever os programas da parte (a) e da parte (b), responda brevemente às seguintes perguntas: Se você não conhecesse a chave para a parte (b), qual seria o grau de dificuldade para quebrar o código? E se você tivesse acesso a substancial poder da computação (por exemplo, supercomputadores)? No Exercício 18.26 solicitamos que você escreva um programa para realizar isso.

**18.8** Escreva um programa com iteradores que demonstra o uso das funções `rbegin` e `rend`.

**18.9** Escreva um programa que lê várias `strings` e imprime somente aquelas que terminam em 'r' ou 'ay'. Somente letras minúsculas devem ser consideradas.

- 18.10** Escreva um programa que demonstra a passagem de uma `string` por referência e por valor.
- 18.11** Escreva um programa que insere um nome e um sobrenome separadamente e concatena os dois em uma nova `string`.
- 18.12** Escreva um programa que reproduz o jogo da forca. O programa deve selecionar uma palavra (que seja codificada diretamente no programa ou lida de um arquivo de texto) e exibir o seguinte:

```
Guess the word: XXXXXX
[Adivinhe a palavra:]
```

Cada X representa uma letra. O usuário tenta adivinhar as letras da palavra. A resposta apropriada `yes` ou `no` deve ser exibida depois de cada suposição. Após cada suposição incorreta, exiba o diagrama com outra parte do corpo preenchida. Após sete suposições incorretas, o usuário deve ser 'enforcado'. A exibição deve ser semelhante a:

```
 0
 /|\
 |
 / \
```

Depois de cada suposição, exiba todas as adivinhações do usuário. Se o usuário adivinhar a palavra corretamente, o programa deve exibir

```
Congratulations!!! You guessed my word. Play again? yes/no
[Parabéns!!! Você adivinhou minha palavra. Quer jogar de novo? sim/não]
```

- 18.13** Escreva um programa que lê uma `string` e a imprime de trás para a frente. Converta todos os caracteres maiúsculos em minúsculos e todos os minúsculos em maiúsculos.
- 18.14** Escreva um programa que utiliza as capacidades de comparação introduzidas neste capítulo para listar em ordem alfabética uma série de nomes de animais. Somente letras maiúsculas devem ser utilizadas para as comparações.
- 18.15** Escreva um programa que cria um criptograma a partir de uma `string`. Um criptograma é uma mensagem ou palavra em que cada letra é substituída por outra. Por exemplo, a `string`

```
The bird was named squawk
```

poderia ser embaralhada para formar

```
cin vrjs otz ethns zxqtop
```

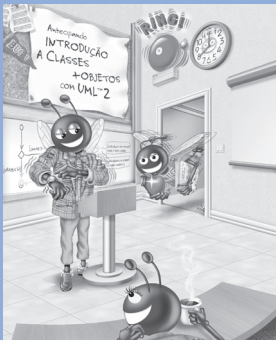
Observe que os espaços não são embaralhados. Nesse caso particular, 'T' foi substituído por 'x', a cada 'a' que foi substituído por 'h' etc. As letras maiúsculas tornam-se minúsculas no criptograma. Utilize técnicas semelhantes àsquelas do Exercício 18.7.

- 18.16** Modifique o Exercício 18.15 para permitir ao usuário resolver o criptograma. O usuário deve inserir dois caracteres por vez: o primeiro caractere especifica uma letra no criptograma e o segundo especifica a letra substituta. Se a letra substituta for correta, substitua a letra no criptograma pela letra substituta em letra maiúscula.
- 18.17** Escreva um programa que insere uma frase e que conta o número de palíndromos nela. Um palíndromo é uma palavra que é lida da mesma maneira da esquerda para a direita e vice-versa. Por exemplo, "tree" não é um palíndromo, mas "noon" é.
- 18.18** Escreva um programa que conta o número total de vogais em uma frase. Gere a saída da frequência de cada vogal.
- 18.19** Escreva um programa que insere os caracteres "\*\*\*\*\*" exatamente no meio de uma `string`.
- 18.20** Escreva um programa que apaga as seqüências "by" e "BY" de uma `string`.
- 18.21** Escreva um programa que insere uma linha de texto, substitui todas as marcas de pontuação por espaços e utiliza a função de biblioteca de `string` do C `strtok` para tokenizar a `string` em palavras individuais.
- 18.22** Escreva um programa que insere uma linha de texto e imprime o texto de trás para a frente. Utilize iteradores na solução.
- 18.23** Escreva uma versão recursiva do Exercício 18.22.
- 18.24** Escreva um programa que demonstra o uso das funções `erase` que aceitam os argumentos `iterator`.
- 18.25** Escreva um programa que gere a exibição que aparece na Figura 18.13, na próxima página, a partir da `string` "abcdefghijklmnopqrstuvwxyz{"
- 18.26** No Exercício 18.7, solicitamos que você escrevesse um algoritmo de criptografia simples. Escreva um programa que tenta decifrar uma mensagem 'rot13' utilizando a substituição de frequência simples. (Suponha que você não conhece a chave.) As letras mais freqüentes na frase criptografada devem ser substituídas por letras mais comumente utilizadas no idioma português (a, e, i, o, u, s, t, r etc.). Escreva as possibilidades para um arquivo. O que torna tão fácil quebrar o código? Como o mecanismo de criptografia pode ser melhorado?
- 18.27** Escreva uma versão da rotina de classificação por seleção (Figura 8.28) que classifica `strings`. Utilize a função `swap` em sua solução.

```
a
bcb
cdedc
defgfed
efghihgfe
fghijkjihgf
ghijklmlkjihg
hijklmnonmlkji
ijklmnopqponmlkji
jklmnopqrsrqponmlkj
klmnopqrstutsrqponmlk
lmnopqrstuvwutsrqponml
mnopqrstuvwxyxwvutsrqponm
nopqrstuvwxyz{zyxwvutsrqpon
```

**Figura 18.13**

- 18.28** Modifique a classe `Employee` das figuras 13.6–13.7 adicionando uma função utilitária `private` chamada `isValidSocialSecurityNumber`. Essa função-membro deve validar o formato de um número de seguro social norte-americano (por exemplo, ###-##-####, onde # é um dígito). Se o formato for válido, retorne `true`; caso contrário, retorne `false`.



*Este é o ar comum que banha o globo.*

Walt Whitman

*Diz-se que a parte mais longa da viagem é a passagem pelo portão.*

Marcos Terêncio Varrão

*Terminais ferroviários... são a porta para a glória e para o desconhecido. Por eles atravessamos para o mundo da aventura e da luz do sol, por eles, infelizmente, retornamos.*

E. M. Forster

*Haverá um momento na vida de um homem que, para chegar aonde precisa — sem portas ou janelas — ele passará por uma parede.*

Bernard Malamud

## Programação Web

### OBJETIVOS

Neste capítulo, você aprenderá:

- O protocolo Common Gateway Interface (CGI).
- O Hypertext Transfer Protocol (HTTP) e cabeçalhos HTTP.
- As funcionalidades do servidor Web.
- O Apache HTTP Server.
- A solicitar documentos de um servidor Web.
- A implementar scripts CGI.
- Como enviar entrada para scripts CGI utilizando formulários XHTML.

- 19.1** Introdução
- 19.2** Tipos de solicitação HTTP
- 19.3** Arquitetura de múltiplas camadas
- 19.4** Acessando servidores Web
- 19.5** Apache HTTP Server
- 19.6** Solicitando documentos XHTML
- 19.7** Introdução à CGI
- 19.8** Transações HTTP simples
- 19.9** Scripts CGI simples
- 19.10** Enviando entrada para um script CGI
- 19.11** Utilizando formulários XHTML para enviar entrada
- 19.12** Outros cabeçalhos
- 19.13** Estudo de caso: uma página Web interativa
- 19.14** Cookies
- 19.15** Arquivos do lado do servidor
- 19.16** Estudo de caso: carrinho de compras
- 19.17** Síntese
- 19.18** Recursos na Internet e na Web

Resumo | Terminologia | Exercícios de revisão | Respostas dos exercícios de revisão | Exercícios

## 19.1 Introdução

Com o advento da World Wide Web, a Internet ganhou imensa popularidade. Isso aumentou significativamente o volume de solicitações feitas por usuários para obter informações de sites Web. Tornou-se evidente que o grau de interatividade entre o usuário e o site Web seria crucial. O poder da Web não reside apenas em servir conteúdo a usuários, mas também em responder a solicitações de usuários e gerar conteúdo Web dinamicamente.

Neste capítulo, discutimos o software especializado — chamado **servidor Web** — que responde a solicitações de clientes (por exemplo, um navegador Web) fornecendo recursos (por exemplo, documentos XHTML<sup>1</sup>) para exibição nesses clientes. Por exemplo, quando um usuário insere um endereço **Uniform Resource Locator (URL)**, como `www.deitel.com`, em um navegador Web, o usuário está solicitando um documento específico a partir de um servidor Web. O servidor Web mapeia o URL para um arquivo no servidor (ou para um arquivo na rede do servidor) e retorna o documento solicitado ao cliente. Durante essa interação, o servidor Web e o cliente se comunicam pelo **Hypertext Transfer Protocol (HTTP)**, um protocolo independente de plataforma para transferir solicitações e arquivos que respondem a essas solicitações por meio da Internet (isto é, entre servidores e navegadores Web).

Nossa discussão de servidor Web introduz o Apache HTTP Server. Para propósitos de ilustração, utilizamos o navegador Web do Microsoft Internet Explorer para solicitar documentos e, posteriormente, exibir o conteúdo retornado de ‘scripts CGI’. Discutimos o Apache HTTP Server e os scripts CGI nas seções 19.5 e 19.7, respectivamente.

## 19.2 Tipos de solicitação HTTP

O HTTP define vários tipos de solicitação (também conhecidas como **métodos de solicitação**), cada um dos quais especifica como um cliente envia uma solicitação para um servidor. Os dois mais comuns são **get** e **post**. Esses tipos de solicitação recuperam e enviam dados de formulário do cliente a partir de e para um servidor Web. Um **formulário** é um elemento XHTML que pode conter campos de texto, botões de opção, caixas de seleção e outros componentes de interface gráfica com o usuário que permitem aos usuários inserir e enviar dados em uma página Web. Os formulários também podem conter campos ocultos, que não são expostos como componentes GUI. Os campos ocultos são utilizados para passar dados adicionais não especificados pelo usuário para o handler de formulário no servidor Web. Você verá exemplos de campos ocultos mais adiante no capítulo.

Uma solicitação **get** obtém (*gets*, ou recupera) informações de um servidor. Essas solicitações freqüentemente recuperam um documento HTML ou uma imagem. Uma solicitação **post** posta (ou envia) dados a um servidor, como informações de autenticação ou dados de um formulário que coleta a entrada de usuário. Normalmente, solicitações **post** são utilizadas para postar uma mensagem

<sup>1</sup> A Extensible HyperText Markup Language (XHTML) substituiu a Hypertext Markup Language (HTML) como a principal forma de descrever conteúdo Web. Os leitores não familiarizados com a XHTML devem consultar o Apêndice J, “Introdução à XHTML”, antes de ler este capítulo.

em um grupo de notícias ou em um fórum de discussão, passam a entrada de usuário para um processo de tratamento de dados e armazenam ou atualizam os dados em um servidor. Uma solicitação `get` envia dados de formulário como parte do URL (por exemplo, `www.searchsomething.com/search?query=consultaDoUsuário`). Nessa solicitação fictícia, as informações que vêm depois do `?` (`query=consultaDoUsuário`) indicam a entrada especificada pelo usuário. Por exemplo, se o usuário realizasse uma pesquisa sobre ‘Massachusetts’, a última parte do URL seria `?query=Massachusetts`. Uma solicitação `get` limita a **string de consulta** (por exemplo, `query=Massachusetts`) a um número predefinido de caracteres que varia de servidor para servidor. Se a string de consulta exceder esse limite, uma solicitação `post` deve ser utilizada.



### Observação de engenharia de software 19.1

*Os dados enviados em uma solicitação `post` não fazem parte do URL e não podem ser vistos por usuários. Os formulários que contêm muitos campos muitas vezes são enviados a servidores Web via solicitações `post`. Campos de formulário sensíveis, como senhas, devem ser enviados utilizando esse tipo de solicitação.*

A Figura 19.1 lista diferentes tipos de solicitação `get` e `post`. Esses métodos não são utilizados com frequência.

Uma solicitação HTTP frequentemente envia dados para um **handler de formulário do lado do servidor** — um programa que reside no servidor Web e é criado por um programador de aplicativos de servidor para tratar solicitações de clientes. Por exemplo, quando um usuário participa de uma pesquisa baseada na Web, o servidor Web recebe as informações especificadas no formulário como parte da solicitação e o handler de formulário processa a pesquisa. Demonstramos como criar handlers de formulário do lado do servidor por todos os exemplos neste capítulo.

Os navegadores frequentemente **armazenam em cache** (isto é, salvam em um disco local) páginas Web para recarregamento rápido, a fim de reduzir a quantidade de dados de que o navegador precisa para fazer download pela Internet. Os navegadores Web frequentemente armazenam em cache as respostas do servidor às solicitações `get`. Uma página Web estática, como o conteúdo curricular de um curso, é armazenada em cache no caso de o usuário solicitar o mesmo recurso novamente. Entretanto, os navegadores normalmente não armazenam em cache as respostas às solicitações `post`, porque as solicitações `post` subsequentes poderiam não conter as mesmas informações. Por exemplo, em uma enquete on-line, muitos usuários podem visitar a mesma página Web e responder a uma pergunta. A página também poderia exibir os resultados de uma enquete. Cada nova resposta altera todos os resultados da enquete, portanto qualquer solicitação para visualizar os resultados de pesquisa deve ser enviada utilizando o método `post`. De maneira semelhante, o método `post` deve ser utilizado para solicitar uma página Web contendo mensagens de um fórum de discussão, uma vez que essas mensagens podem mudar frequentemente. Caso contrário, o navegador pode armazenar os resultados em cache depois da primeira visita do usuário e exibir esses mesmos resultados para cada visita subsequente.

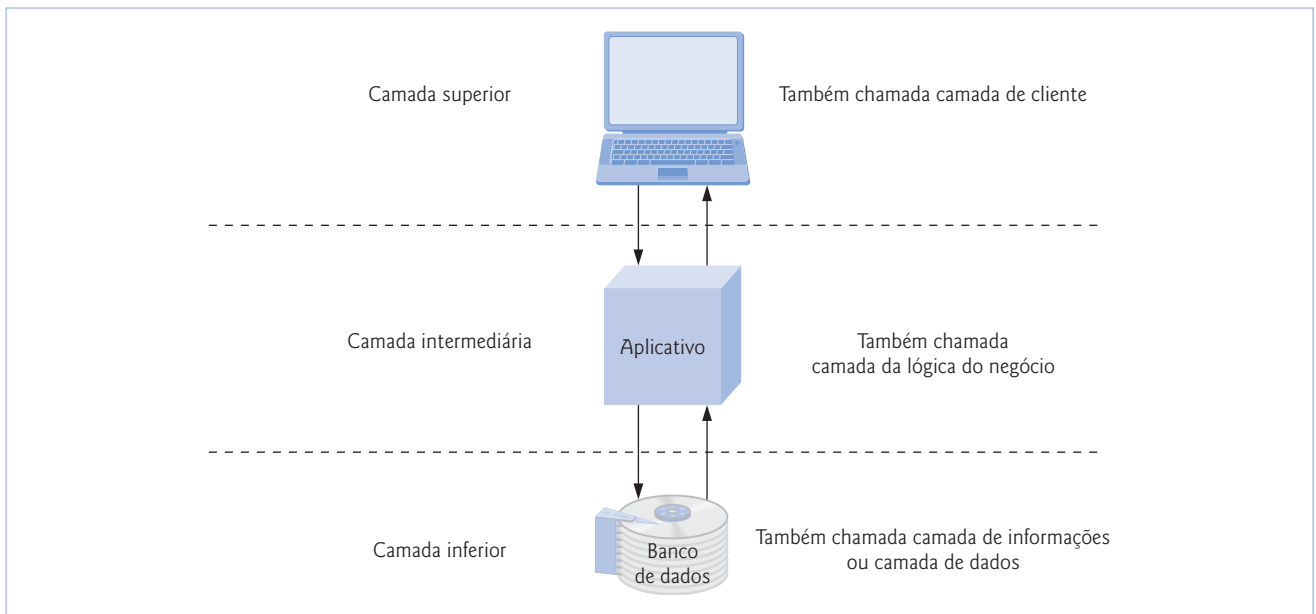
## 19.3 Arquitetura de múltiplas camadas

Um servidor Web faz parte de um **aplicativo de múltiplas camadas**, às vezes referido como um **aplicativo de  $n$  camadas**. Os aplicativos de múltiplas camadas dividem as funcionalidades em **camadas** separadas (isto é, agrupamentos lógicos de funcionalidade). As camadas podem ser localizadas no mesmo computador ou em computadores separados. A Figura 19.2 apresenta a estrutura básica de um aplicativo de três camadas.

| Tipo de solicitação  | Descrição                                                                                                                                                                                                                                                                                                                                                                 |
|----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>delete</code>  | Esta solicitação é normalmente utilizada para excluir um arquivo de um servidor. Isso talvez não esteja disponível em alguns servidores por causa dos riscos inerentes de segurança (por exemplo, o cliente poderia excluir um arquivo que é crítico para a execução do servidor ou para um aplicativo).                                                                  |
| <code>head</code>    | Normalmente, esta solicitação é utilizada quando o cliente quer somente os cabeçalhos da resposta, como o tipo do seu conteúdo e comprimento do conteúdo.                                                                                                                                                                                                                 |
| <code>options</code> | Este tipo de solicitação retorna informações ao cliente indicando as opções de HTTP suportadas pelo servidor, como a versão do HTTP (1.0 ou 1.1) e os métodos de solicitação que o servidor suporta.                                                                                                                                                                      |
| <code>put</code>     | Esta solicitação é normalmente utilizada para armazenar um arquivo no servidor. Isso talvez não esteja disponível em alguns servidores por causa dos riscos inerentes de segurança (por exemplo, o cliente poderia colocar um aplicativo executável no servidor, que, se executado, danificaria o servidor — talvez excluindo arquivos críticos ou ocupando os recursos). |
| <code>trace</code>   | Esta solicitação é normalmente utilizada para depuração. A implementação desse método retorna automaticamente um documento XHTML para o cliente contendo as informações de cabeçalho da solicitação (dados enviados pelo navegador como parte da solicitação).                                                                                                            |

**Figura 19.1** Outros tipos de solicitação HTTP.





**Figura 19.2** Modelo de aplicativo de três camadas.

A **camada inferior** (também chamada de **camada de informações** ou **camada de dados**) mantém os dados do aplicativo. Essa camada, em geral, armazena os dados em um **sistema de gerenciamento de banco de dados relacional** (*relational database management system – RDBMS*). Por exemplo, uma loja de varejo poderia ter um banco de dados de informações de produto, como descrições, preços e quantidades de itens no estoque. O mesmo banco de dados também poderia conter informações sobre clientes, como nomes de usuário para efetuar login na loja on-line, endereços de cobrança e números de cartão de crédito.

A **camada intermediária** (também chamada de **camada da lógica do negócio**) implementa a **lógica do negócio** para controlar interações entre clientes de aplicativos e dados de aplicativo. Ela atua como um intermediário entre os dados na camada de informações e os clientes do aplicativo. A lógica do negócio da camada intermediária recebe solicitações de clientes a partir da camada superior (por exemplo, uma solicitação para ver um catálogo de produtos), recupera e processa dados da camada de informações e envia o resultado de volta para o cliente.

A lógica do negócio na camada intermediária impõe **regras de negócio** e assegura que os dados são confiáveis antes de atualizar o banco de dados ou enviar os dados para um usuário. As regras de negócio determinam como os clientes podem e não podem acessar dados de aplicativo e como os aplicativos processam os dados. Por exemplo, uma regra de negócio pode especificar como converter notas de avaliação numéricas em notas baseadas em letras.

A **camada superior** (também chamada de **camada de cliente**) é a interface com o usuário do aplicativo. Os usuários interagem diretamente com o aplicativo por meio da interface com o usuário. A camada de cliente interage com a camada intermediária para fazer solicitações e recuperar dados da camada de informações. A camada de cliente então exibe para o usuário os dados recuperados da camada intermediária.

## 19.4 Acessando servidores Web

Para solicitar documentos de servidores Web, os usuários devem conhecer os URLs em que esses documentos residem. Os usuários podem solicitar documentos de **servidores Web locais** (isto é, aqueles que residem nas máquinas dos usuários) ou **servidores Web remotos** (isto é, aqueles que residem em máquinas por meio de uma rede).

Para entender como um navegador Web é capaz de localizar documentos em um servidor Web, é útil conhecer os seguintes termos:

1. **Host**: Um **host** é um computador que armazena e mantém recursos, como páginas Web, bancos de dados e arquivos multimídia.
2. **Domínio**: Um **domínio** representa um grupo de hosts na Internet. Cada domínio tem um **nome de domínio**, também conhecido como um **endereço Web**, que identifica de maneira exclusiva a localização de um negócio ou organização na Internet.
3. **Nome de domínio completamente qualificado**: Um **nome de domínio completamente qualificado** (*fully qualified domain name – FQDN*), também conhecido como o nome de máquina, contém um host (por exemplo, **www** para World Wide Web) e um nome de domínio, incluindo um **domínio de nível mais alto** (*top-level domain – TLD*). O domínio de nível mais alto é o último e mais significativo componente de um nome de domínio completamente qualificado.

Os servidores Web locais podem ser acessados de duas maneiras: pelo nome de máquina ou pelo **localhost** — um nome de host que referencia a máquina local. Utilizamos **localhost** neste livro para propósitos de demonstração. Para determinar o nome de máquina no Windows 98, clique com o botão direito do mouse em Ambiente de Rede e selecione Propriedades a partir do menu de contexto para exibir a caixa de diálogo Rede. Na caixa de diálogo Rede, clique na guia Identificação. O nome do computador aparece no campo Nome de computador. Clique em Cancel para fechar a caixa de diálogo Rede. No Windows 2000, clique com o botão direito do mouse em Meus Locais de Rede e selecione Propriedades no menu de contexto para exibir a janela Explorer Conexões de Rede e Dialup. Na janela Explorer, clique em Identificação de rede. O campo Nome completo do computador: na janela Propriedades de sistema exibe o nome do computador. No Windows XP, selecione Iniciar > Painel de Controle, que exibe a janela Painel de Controle. Dê um clique duplo em Sistema na janela Painel de Controle, que abre a janela Propriedades de sistema. Selecione a guia Nome de computador na janela Propriedades de sistema; o campo Nome completo de computador: exibe o nome de computador.

Para solicitar documentos do servidor Web, os usuários devem conhecer os nomes de domínio completamente qualificados (nomes de máquina) em que o software de servidor Web reside. Por exemplo, para acessar os documentos de servidor Web da Deitel, você deve conhecer o FQDN `www.deitel.com`. O FQDN `www.deitel.com` indica que o host é `www` e o domínio de nível mais alto é `com`. Em um FQDN, o TLD frequentemente descreve o tipo de organização que possui o domínio. Por exemplo, normalmente o TLD `com` refere-se a um negócio comercial, o TLD `org` a uma organização sem fins lucrativos e o TLD `edu` a uma instituição educacional. Além disso, cada país tem seu próprio TLD, como `cn` para China, `et` para Etiópia, `om` para Omã e `us` para os Estados Unidos.

Cada FQDN corresponde a um endereço numérico chamado **endereço IP (Internet Protocol)**, que é muito semelhante a um endereço de rua. Assim como as pessoas utilizam endereços de rua para encontrar casas ou empresas em uma cidade, os computadores utilizam endereços IP para localizar outros computadores na Internet. Cada computador host da internet tem um endereço IP único. Cada endereço abrange quatro conjuntos de números separados por pontos, como `63.110.43.82`. Um **servidor DNS (Domain Name System)** é um computador que mantém um banco de dados FQDNs e seus endereços IP correspondentes. O processo de traduzir FQDNs em endereços IP é chamado **pesquisa DNS**. Por exemplo, para acessar o site Web Deitel, digite o FQDN `www.deitel.com` em um navegador Web. A pesquisa DNS traduz `www.deitel.com` no endereço IP do servidor Web da Deitel (`63.110.43.82`). O endereço IP de `localhost` é sempre `127.0.0.1`. Esse endereço, também conhecido como o **endereço de loopback**, pode ser utilizado para testar aplicativos Web em seu computador local.

## 19.5 Apache HTTP Server

O **Apache HTTP Server**, mantido pela Apache Software Foundation, é atualmente o servidor Web mais popular por causa de sua estabilidade, custo, eficiência e portabilidade. É um software de **código-fonte aberto** (isto é, seu código-fonte está livremente disponível) que executa em plataformas UNIX, Linux e Windows.

Para fazer download do Apache HTTP Server, visite `httpd.apache.org`.<sup>2</sup> Para obter instruções sobre a instalação do Apache, visite `www.deitel.com` ou `httpd.apache.org`. Se o Apache HTTP Server estiver instalado como um serviço, então ele já está executando depois da instalação. Caso contrário, inicie o servidor selecionando o menu Iniciar e, então, Todos os programas > Apache HTTP Server 2.0.52 > Control Apache Server > Start. Para parar o Apache HTTP Server, selecione Iniciar > Todos os programas > Apache HTTP Server 2.0.52 > Control Apache Server > Stop. Para usuários do Linux, publicamos instruções sobre como iniciar/parar o Apache HTTP Server e executar os exemplos em nosso site Web, `www.deitel.com`.

## 19.6 Solicitando documentos XHTML

Esta seção mostra como solicitar um documento XHTML a partir do Apache HTTP Server. Na estrutura de diretório de Apache HTTP Server, os documentos XHTML devem ser salvos no diretório **htdocs**. Em plataformas Windows, o diretório `htdocs` reside em `C:\Arquivos de programas\Apache Group\Apache2`. Em plataformas Linux, o diretório `htdocs` reside no diretório `/usr/local/httpd`.<sup>3</sup> Copie o documento `test.html` do diretório de exemplos do Capítulo 19 do CD-ROM. Para solicitar o documento, carregue um navegador Web, como o Internet Explorer ou o Netscape, e insira o URL no campo Endereço (isto é, `http://localhost/test.html`). A Figura 19.3 mostra o resultado de solicitar `test.html`. [Nota: No Apache, o diretório-raiz do servidor Web refere-se ao diretório-padrão, `htdocs`, então não inserimos o nome de diretório antes do nome de arquivo (isto é, `test.html`) no campo Endereço.]

## 19.7 Introdução à CGI

O **Common Gateway Interface (CGI)** é um protocolo-padrão para permitir aos aplicativos (comumente chamados **programas CGI** ou **scripts CGI**) interagir com servidores Web e (indiretamente) com clientes (por exemplo, navegadores Web). O CGI é frequentemente utilizado para gerar **conteúdo Web dinâmico** utilizando a entrada de cliente, bancos de dados e outros serviços de informações. Uma página Web é dinâmica se seu conteúdo é gerado programaticamente quando a página é solicitada, ao contrário do **conteúdo Web estático**, que não é gerado programaticamente quando a página é solicitada (isto é, a página já existe antes de a solicitação ser feita). Por exemplo, podemos utilizar uma página Web estática a fim de solicitar que um usuário insira um CEP e, então, redirecionar o usuário para

<sup>2</sup> Neste capítulo, utilizamos a versão 2.0.52.

<sup>3</sup> Os usuários Linux já podem ter Apache instalado por padrão. O diretório `htdocs` pode ser localizado em vários lugares, dependendo da distribuição do Linux.

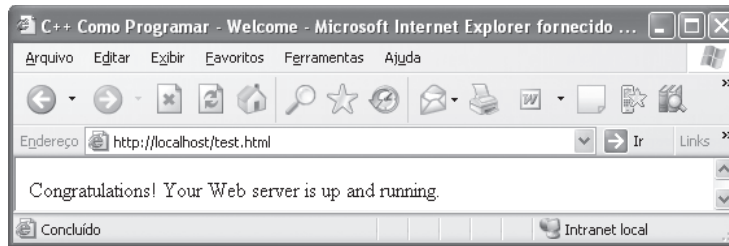


Figura 19.3 Solicitando test.html a partir do Apache.

um script CGI que gera uma página Web dinâmica personalizada para as pessoas dessa área geográfica. Neste capítulo, introduzimos os princípios básicos de CGI e utilizamos o C++ para escrever nossos primeiros scripts CGI.

O Common Gateway Interface é ‘comum’ no sentido de que não é específico a nenhum sistema operacional particular (como Linux ou Windows) ou a nenhuma linguagem de programação. O CGI foi projetado para ser utilizado com praticamente todas as linguagens de programação, como C, C++, Perl, Python ou Visual Basic.

O CGI foi desenvolvido em 1993 pelo **NCSA (National Center for Supercomputing Applications — [www.ncsa.uiuc.edu](http://www.ncsa.uiuc.edu))** para ser utilizado com seu popular **servidor Web HTTPd**. Diferentemente dos protocolos Web e linguagens que têm especificações formais, a concisa descrição inicial de CGI escrita pelo NCSA provou-se tão simples que o CGI foi adotado não oficialmente como um padrão mundial. O suporte CGI foi incorporado rapidamente em outros servidores Web, incluindo o Apache.

## 19.8 Transações HTTP simples

Antes de explorar como o CGI opera, é necessário ter um entendimento básico de rede e de como a World Wide Web funciona. Nesta seção, examinamos o funcionamento interno do *Hypertext Transfer Protocol (HTTP)* e discutimos o que acontece nos bastidores quando um navegador faz uma solicitação e, então, exibe a resposta. O HTTP descreve um conjunto de **métodos** e **cabeçalhos** que permite aos clientes e servidores interagir e trocar informações de maneira uniforme e previsível.

Uma página Web na sua forma mais simples é um documento XHTML, que é um arquivo de texto simples contendo marcações (**elementos**) que descrevem a estrutura dos dados que o documento contém. Por exemplo, o XHTML

```
<title>My Web Page</title>
```

indica para o navegador que o texto entre o **tag inicial <title>** e o **tag final </title>** é o título da página Web. Os documentos XHTML também podem conter informações de **hipertexto** (normalmente chamadas de **hyperlinks**), que são vinculadas com outras páginas Web ou outras localizações na mesma página. Quando um usuário ativa um hyperlink (normalmente clicando nele com o mouse), o navegador Web ‘segue’ o hyperlink carregando a nova página Web (ou uma parte diferente da mesma página Web) a partir do servidor Web que contém a página Web.

Cada arquivo XHTML disponível para visualização pela Web tem um URL associado com ele. Um URL contém o protocolo do recurso (como http), o nome de máquina ou endereço IP do recurso e o nome (incluindo o caminho) do recurso. Por exemplo, no URL

```
http://www.deitel.com/books/downloads.html
```

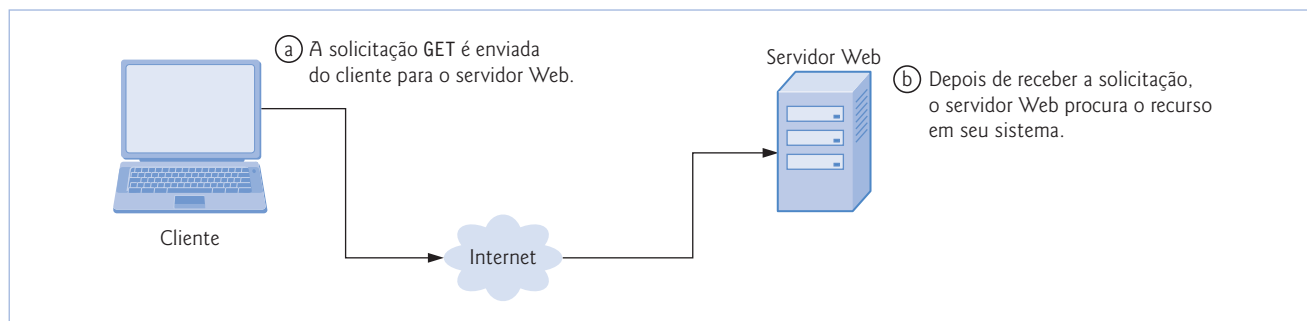
o protocolo é http, o nome de máquina é www.deitel.com. O nome do recurso sendo solicitado, /books/downloads.html (um documento XHTML), é o restante do URL. Essa parte do URL especifica tanto o nome do recurso (downloads.html) como seu caminho (/books), o que ajuda o servidor Web a processar a solicitação para determinar onde o recurso está localizado no servidor Web. Observe que um documento XHTML termina com a extensão de arquivo .html. O caminho poderia representar um diretório real no sistema de arquivos do servidor Web. Entretanto, por razões de segurança, o caminho muitas vezes refere-se a um **diretório virtual** — um alias ou nome falso de um diretório físico em disco. Nesse caso, o servidor converte o caminho em uma localização real no servidor (ou até mesmo em outro computador), ocultando, assim, a verdadeira localização do recurso. De fato, é até possível que o recurso seja criado dinamicamente e não resida em nenhum lugar do computador do servidor. Como veremos, os URLs também podem ser utilizados para especificar a entrada do usuário para um programa no servidor.

Agora consideramos como um navegador, ao receber um URL, realiza uma transação HTTP simples para recuperar e exibir uma página Web. A Figura 19.4 ilustra a transação em detalhes. A transação é realizada entre um navegador Web e um servidor Web.

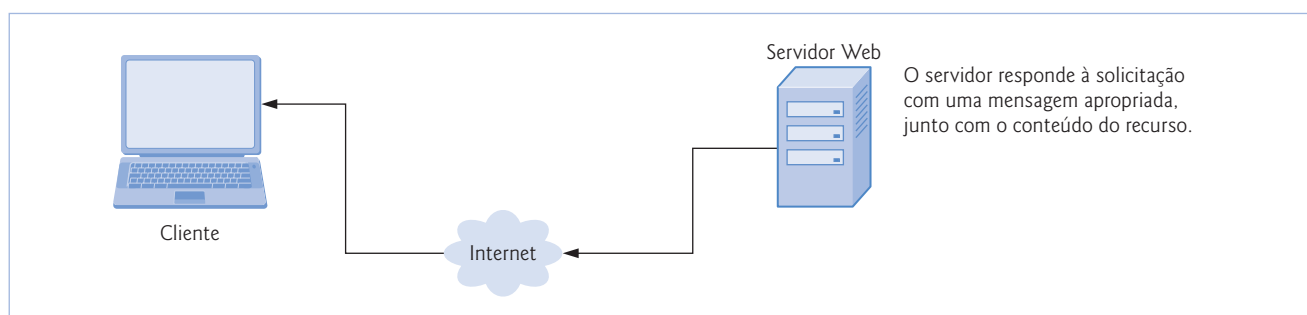
No Passo 1 da Figura 19.4a, o navegador envia uma solicitação HTTP para o servidor. A solicitação (em sua forma mais simples) é semelhante a

```
GET /books/downloads.html HTTP/1.1
Host: www.deitel.com
```

A palavra **GET**, um método de HTTP, indica que o cliente envia uma solicitação get e deseja recuperar um recurso. O restante da solicitação fornece o nome e o caminho do recurso (/books/downloads.html) e o nome do protocolo e número de versão (HTTP/1.1). Depois de o servidor Web receber a solicitação, ele pesquisa o recurso pelo sistema.



**Figura 19.4a** Interação de cliente com servidor e servidor Web. Passo 1: A solicitação get, GET /books/downloads.html HTTP/1.1.



**Figura 19.4b** Interação de cliente com servidor e servidor Web. Passo 2: A resposta de HTTP, HTTP/1.1 200 OK.

Qualquer servidor que entende HTTP (versão 1.1) será capaz de traduzir essa solicitação e responder apropriadamente. O Passo 2 da Figura 19.4b mostra os resultados de uma solicitação bem-sucedida. O servidor primeiro envia uma resposta que indica a versão de HTTP, seguida por um código numérico e uma frase que descreve o status da transação. Por exemplo,

```
HTTP/1.1 200 OK
```

indica sucesso;

```
HTTP/1.1 404 Not found
```

informa o cliente que o recurso solicitado não foi encontrado no servidor na localização especificada.

O servidor então envia um ou mais cabeçalhos HTTP, que fornecem informações sobre os dados sendo enviados para o cliente. Nesse caso, o servidor está enviando um documento XHTML, então o cabeçalho HTTP exhibe

```
Content-Type: text/html
```

As informações no **cabeçalho Content-Type** identificam o **tipo MIME** (*Multipurpose Internet Mail Extensions*) do conteúdo. Cada documento do servidor tem um tipo MIME pelo qual o navegador determina como processar os dados que ele recebe. Por exemplo, o tipo MIME **text/plain** indica que os dados contêm texto que deve ser exibido sem tentar interpretar nenhum conteúdo como marcação XHTML. De maneira semelhante, o tipo MIME **image/gif** indica que o conteúdo é uma imagem GIF. Quando esse tipo MIME for recebido pelo navegador, ele tentará exibir os dados como uma imagem.

Os cabeçalhos são seguidos por uma linha em branco, o que indica para o cliente que o servidor terminou de enviar os cabeçalhos HTTP. O servidor então envia o conteúdo do documento solicitado (por exemplo, `downloads.html`). A conexão termina quando a transferência do recurso estiver completa (nesse caso, quando o fim do documento `downloads.html` é alcançado). O navegador do lado do cliente interpreta a XHTML que ele recebe e renderiza (ou exibe) os resultados.

## 19.9 Scripts CGI simples

Contanto que um arquivo XHTML no servidor permaneça inalterado, seu URL associado exibirá o mesmo conteúdo em navegadores dos clientes toda vez que o arquivo for acessado. Para alterar o conteúdo do arquivo XHTML (por exemplo, incluindo novos links ou as últimas notícias da empresa), alguém deve alterar manualmente o arquivo no servidor, provavelmente com um editor de textos ou software de design de página Web.

Essa necessidade de alteração manual é um problema para os autores de páginas Web que querem criar páginas Web dinâmicas interessantes. Fazer uma pessoa alterar continuamente uma página Web é tedioso. Por exemplo, se você quiser que sua página Web exiba sempre a data atual ou condições de clima, a página exigirá atualização contínua.

*Primeiro script CGI*

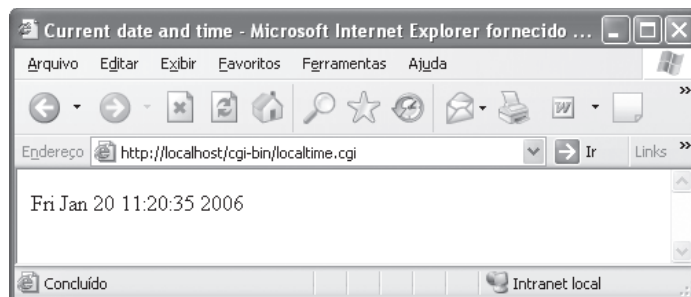
A Figura 19.5 mostra nosso primeiro script CGI. Observe que o programa consiste principalmente em instruções `cout` (linhas 16–29). Até agora, a saída de `cout` sempre foi exibida na tela. Entretanto, tecnicamente falando, o alvo-padrão para `cout` é a saída-padrão. Quando um programa C++ é executado como um script CGI, a saída-padrão é redirecionada pelo servidor Web para o navegador Web do cliente. Para executar o programa como um script CGI, colocamos o arquivo executável C++ compilado no diretório `cgi-bin` do servidor Web. Para o propósito deste capítulo, mudamos a extensão de arquivo executável de `.exe` para `.cgi`.<sup>4</sup> Supondo que o servidor Web esteja em seu computador local, você pode executar o script digitando

```
http://localhost/cgi-bin/localtime.cgi
```

```

1 // Figura 19.5: localtime.cpp
2 // Exibe a data e hora atual em um navegador Web.
3 #include <iostream>
4 using std::cout;
5
6 #include <ctime> // definições de time_t, time, localtime e asctime
7 using std::time_t;
8 using std::time;
9 using std::localtime;
10 using std::asctime;
11
12 int main()
13 {
14 time_t currentTime; // variável para armazenar time
15
16 cout << "Content-Type: text/html\n\n"; // gera saída do cabeçalho HTTP
17
18 // gera saída da declaração XML e DOCTYPE
19 cout << "<?xml version = \"1.0\"?>"
20 << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.1//EN\" "
21 << "\"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd\">";
22
23 time(¤tTime); // armazena time em currentTime
24
25 // gera saída do elemento html e parte de seu conteúdo
26 cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
27 << "<head><title>Current date and time</title></head>"
28 << "<body><p>" << asctime(localtime(¤tTime))
29 << "</p></body></html>";
30 return 0;
31 } // fim de main

```



**Figura 19.5** Primeiro script CGI.

<sup>4</sup> Em um servidor executando Windows Microsoft, o executável pode ser executado diretamente na forma `.exe`.



no campo Endereço ou Localização do seu navegador. Se estiver solicitando esse script a partir de um servidor Web remoto, você precisará substituir `localhost` pelo nome de máquina do servidor ou endereço IP.

A noção de saída-padrão é semelhante àquela da entrada-padrão, que vimos associada com `cin`. Assim como a entrada-padrão se refere à origem-padrão de entrada em um programa (normalmente, o teclado), a saída-padrão refere-se ao destino-padrão de saída de um programa (normalmente, a tela).

É possível redirecionar (ou fazer *pipe*) a saída-padrão para outro destino. Portanto, em nosso script CGI, quando geramos saída de um cabeçalho de HTTP (linha 16) ou elementos XHTML (linhas 19–21 e 26–29), a saída é enviada para o servidor Web, em oposição à tela. O servidor envia essa saída para o cliente pelo HTTP, que interpreta os cabeçalhos e elementos como se eles fizessem parte de uma resposta de servidor normal a uma solicitação de documento XHTML.

É relativamente simples e direto escrever um programa C++ que gera saída da data e hora atual (para o monitor do computador local). De fato, isso requer somente algumas linhas de código (linhas 14, 23 e 28). A linha 14 declara `currentTime` como uma variável de tipo `time_t`. A função `time` (linha 23) obtém a hora atual, que é representada como o número de segundos passados desde a meia-noite de 1º de janeiro de 1970 e armazena o valor recuperado para a localização especificada pelo parâmetro (nesse caso, `currentTime`). A função `localtime` de biblioteca C++ (linha 28), quando recebe uma variável `time_t` (por exemplo, `currentTime`), retorna um ponteiro para um objeto contendo a hora local separada em campos (isto é, dias, horas etc. são colocados em variáveis-membro individuais). A função `asctime` (linha 28), que aceita um ponteiro para um objeto contendo a hora dividida em campos, retorna uma string como

```
Wed Oct 31 13:10:37 2004
```

E se desejássemos enviar a hora atual para a janela de navegador de um cliente para exibir (em vez de gerar sua saída na tela)? O CGI torna isso possível redirecionando a saída de um programa para o próprio servidor Web, que então envia a saída para o navegador de um cliente.

### *Como o servidor Web redireciona a saída*

A Figura 19.6 ilustra esse processo em mais detalhes. No Passo 1, o cliente solicita o recurso chamado `localtime.cgi` a partir do servidor, assim como solicitou `downloads.html` no exemplo anterior (Figura 19.4). Se o servidor não foi configurado para tratar scripts CGI, ele poderia simplesmente retornar o conteúdo do arquivo executável C++ ao cliente, como se fosse qualquer outro documento. Entretanto, com base na configuração do servidor Web, o servidor executa `localtime.cgi` (implementado com o C++) e envia a saída do script CGI para o navegador Web.

Entretanto, um servidor Web adequadamente configurado reconhecerá que diferentes tipos de recursos devem ser tratados diferentemente. Por exemplo, quando o recurso for um script CGI, o script deve ser executado pelo servidor antes de ser enviado. Um script CGI é designado em uma de duas maneiras: ele tem uma extensão do nome do arquivo especial (como `.cgi` ou `.exe`) ou é localizado em um diretório específico (frequentemente `cgi-bin`). Além disso, o administrador de servidor deve explicitamente dar permissão para que os clientes remotos sejam capazes de acessar e executar os scripts CGI.<sup>5</sup>

No Passo 2 da Figura 19.6b, o servidor reconhece que o recurso é um script CGI e executa o script. No Passo 3, a saída produzida pelas três instruções `cout` do script (linhas 16, 19–21 e 26–29 da Figura 19.5) é enviada para a saída-padrão e retornada ao servidor Web. Por fim, no Passo 4, o servidor Web adiciona uma mensagem à saída que indica o status da transação HTTP (como HTTP/1.1 200 OK, para sucesso) e envia a saída inteira do programa CGI para o cliente.

O navegador do lado do cliente então processa o documento XHTML e exibe os resultados. É importante observar que o navegador ignora o que acontece no servidor. Em outras palavras, no que diz respeito ao navegador, ele solicita um recurso como qualquer outro e recebe uma resposta como qualquer outra. O navegador recebe e interpreta a saída do script exatamente como se ela fosse um documento XHTML estático simples.

De fato, você pode visualizar o conteúdo que o navegador recebe executando `localtime.cgi` a partir da linha de comando, como normalmente executaríamos qualquer programa dos capítulos anteriores. [Nota: A extensão do nome do arquivo deve ser mudada para `.exe` antes de executá-lo a partir da linha de comando em um sistema executando Windows.] A Figura 19.7 mostra a saída. Para o propósito deste capítulo, formatamos a saída por uma questão de legibilidade. Note que, com o script CGI, devemos gerar saída do cabeçalho `Content-Type`, enquanto, para um documento XHTML, o servidor Web incluiria o cabeçalho.

O script CGI imprime o cabeçalho `Content-Type`, uma linha em branco e os dados (XHTML, texto simples etc.) para a saída-padrão. Quando o script CGI é executado no servidor Web, este recupera a saída do script, insere a resposta de HTTP no início e entrega o conteúdo para o cliente.

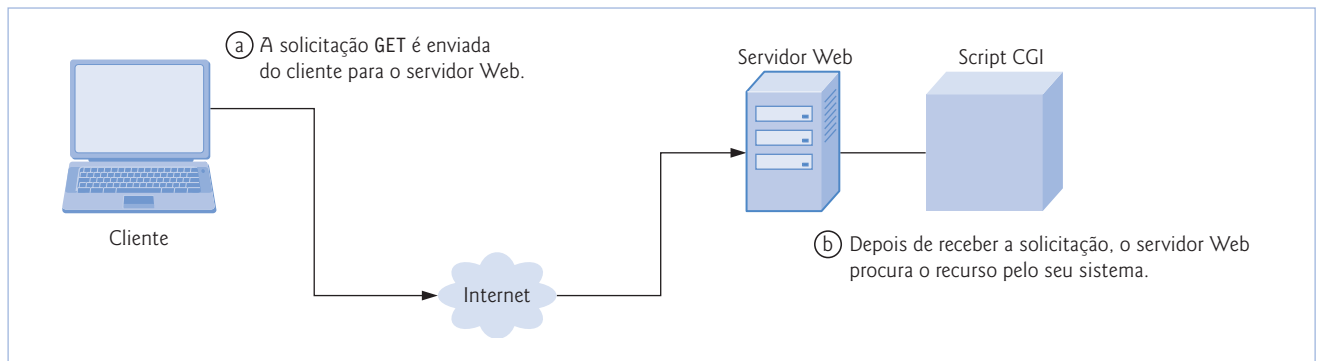
Mais adiante veremos outros tipos de conteúdo que podem ser utilizados dessa maneira, bem como outros cabeçalhos que podem ser utilizados além de `Content-Type`.



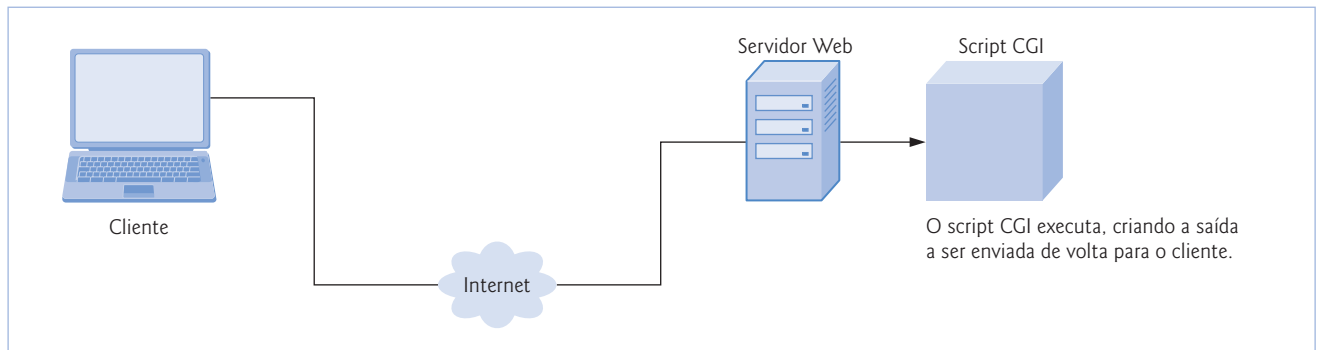
### **Erro comum de programação 19.1**

*Esquecer de colocar uma linha em branco depois de um cabeçalho é um erro de sintaxe.*

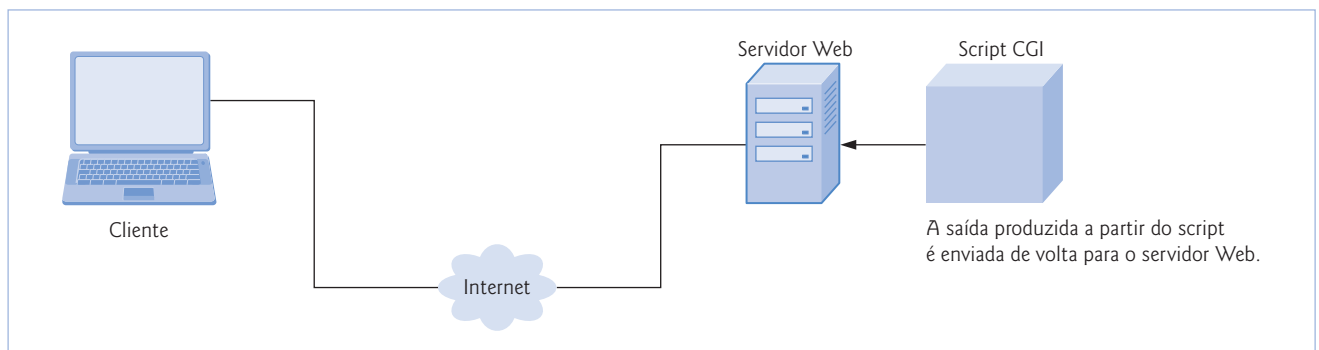
<sup>5</sup> Se estiver utilizando o servidor Apache HTTP e quiser obter informações adicionais sobre a configuração, consulte a home page do Apache em <http://apache.org/docs-2.0/>.



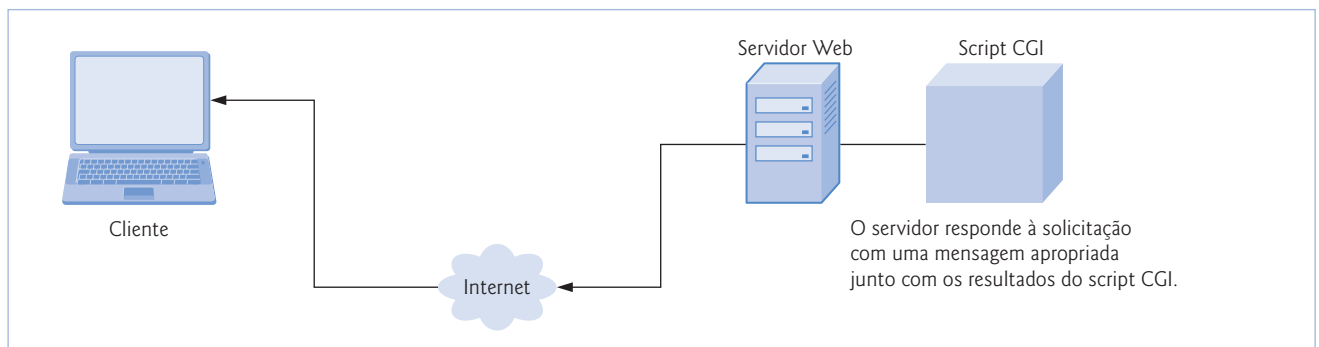
**Figura 19.6a** Passo 1: A solicitação get, GET /cgi-bin/localtime.cgi HTTP/1.1.



**Figura 19.6b** Passo 2: O servidor Web inicia o script CGI.



**Figura 19.6c** Passo 3: A saída de script é enviada para o servidor Web.



**Figura 19.6d** Passo 4: A resposta de HTTP, HTTP/1.1 200 OK.



```
Content-Type: text/html

<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<html xmlns = "http://www.w3.org/1999/xhtml">
 <head>
 <title>Current date and time</title>
 </head>

 <body>
 <p>Wed Oct 13 10:22:18 2004</p>
 </body>
</html>
```

**Figura 19.7** A saída de `localtime.cgi` quando executada a partir da linha de comando.

### Exibindo variáveis de ambiente

O programa da Figura 19.8 gera saída das **variáveis de ambiente** que o Apache HTTP Server configura para scripts CGI. Essas variáveis contêm informações sobre o cliente e o ambiente de servidor, como o tipo de navegador Web utilizado e a localização de um documento no servidor. As linhas 14–23 inicializam um array de objetos `string` com os nomes das variáveis de ambiente CGI. [Nota: As variáveis de ambiente são específicas do servidor. Os servidores diferentes do Apache HTTP Server podem não fornecer todas essas variáveis de ambiente.] A linha 37 inicia a tabela XHTML em que os dados serão exibidos.

As linhas 42–52 geram saída de cada linha da tabela. Vamos examinar minuciosamente cada uma dessas linhas. A linha 42 gera saída de um tag inicial XHTML `<tr>` (linha de tabela), que indica o começo de uma nova linha de tabela. A linha 52 gera saída de um tag final `</tr>` correspondente, que indica o fim da linha. Cada linha da tabela contém duas células de tabela — o nome de uma variável de ambiente e os dados associados com essa variável (se a variável existir). O tag inicial `<td>` (linha 42) começa em uma nova célula de tabela. O loop `for` (linhas 40–53) itera por cada um dos 24 objetos `string`. A saída do nome de cada variável de ambiente é gerada na célula esquerda da tabela (linha 42). A linha 45 tenta recuperar o valor associado com a variável de ambiente chamando a função `getenv` de `<stdlib>` e passando para ela o valor de `string` retornado a partir da chamada de função `environmentVariables[ i ].c_str()`. A função `c_str` retorna uma `string char *` no estilo C contendo o conteúdo da `environmentVariables[ i ] string`. A função `getenv` retorna uma `string char *` contendo o valor de uma variável de ambiente especificada ou um ponteiro nulo se a variável de ambiente não existir.

As linhas 47–50 geram saída do conteúdo da célula direita da tabela. Se a variável de ambiente atual existir (isto é, `getenv` não retornou um ponteiro nulo), a linha 48 gera saída do `value` retornado pela função `getenv`. Se a variável de ambiente não existir no servidor que executa o script, a linha 50 gera saída de uma mensagem apropriada. A execução de amostra mostrada na Figura 19.8 foi produzida executando esse exemplo no Apache HTTP Server, portanto a saída contém dados associados a cada uma das variáveis de ambiente. Os resultados em outros servidores podem variar. Por exemplo, se você fosse executar esse exemplo no Internet Information Services (IIS) da Microsoft, várias das células de tabela na coluna direita exibiriam a mensagem "Environment variable does not exist".

## 19.10 Enviando entrada para um script CGI

Embora as variáveis de ambiente predefinidas forneçam muitas informações, gostaríamos de ser capazes de fornecer qualquer tipo de dados para nossos scripts CGI, como o nome de um usuário ou uma consulta de sistema de pesquisa. A variável de ambiente **QUERY\_STRING** fornece um mecanismo para fazer precisamente isso. A variável `QUERY_STRING` contém informações que são acrescentadas a um URL em uma solicitação `get`. Por exemplo, o URL

```
www.algumsite.com/cgi-bin/script.cgi?state=California
```

faz com que o navegador Web solicite um script CGI (`cgi-bin/script.cgi`) com a string de consulta (`state=California`) a partir de `www.algumsite.com`. O servidor Web armazena a string de consulta depois do `?` na variável de ambiente `QUERY_STRING`. A string de consulta fornece parâmetros que personalizam a solicitação para um cliente particular. Observe que o ponto de interrogação (`?`) não faz parte do recurso solicitado, nem da string de consulta. Serve como um delimitador (ou separador) entre os dois.

A Figura 19.9 mostra um exemplo simples de script CGI que lê os dados passados pelo `QUERY_STRING`. Os dados na string de consulta podem ser formatados de várias maneiras. O script CGI que lê a string de consulta deve saber como interpretar os dados formatados. No exemplo da Figura 19.9, a string de consulta contém uma série de pares nome-valor delimitada pelo *e* comercial (`&`), como em `name=Jill&age=22`.

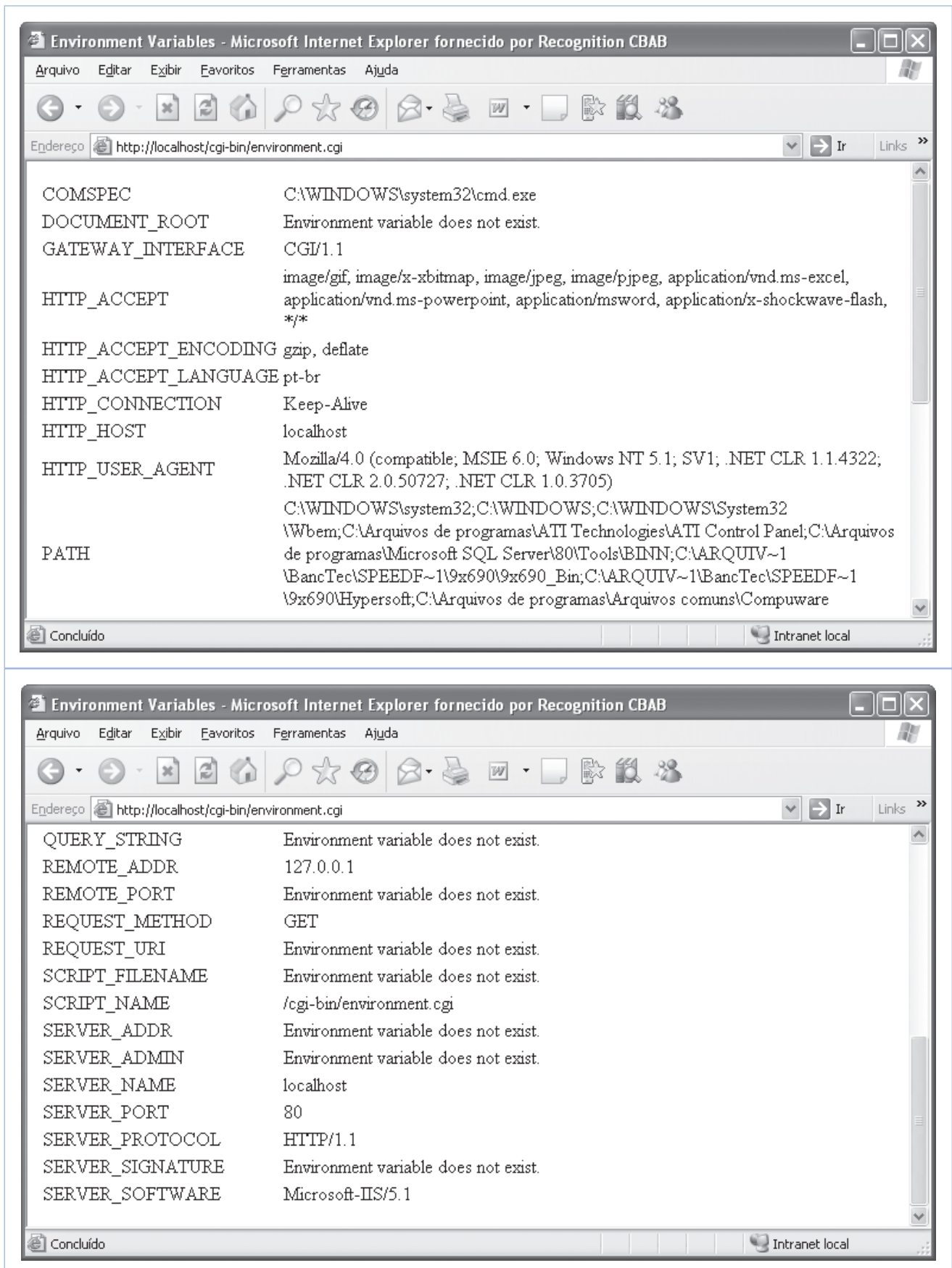
```

1 // Figura 19.8: environment.cpp
2 // Programa para exibir variáveis de ambiente CGI.
3 #include <iostream>
4 using std::cout;
5
6 #include <string>
7 using std::string;
8
9 #include <cstdlib>
10 using std::getenv;
11
12 int main()
13 {
14 string environmentVariables[24] = {
15 "COMSPEC", "DOCUMENT_ROOT", "GATEWAY_INTERFACE",
16 "HTTP_ACCEPT", "HTTP_ACCEPT_ENCODING",
17 "HTTP_ACCEPT_LANGUAGE", "HTTP_CONNECTION",
18 "HTTP_HOST", "HTTP_USER_AGENT", "PATH",
19 "QUERY_STRING", "REMOTE_ADDR", "REMOTE_PORT",
20 "REQUEST_METHOD", "REQUEST_URI", "SCRIPT_FILENAME",
21 "SCRIPT_NAME", "SERVER_ADDR", "SERVER_ADMIN",
22 "SERVER_NAME", "SERVER_PORT", "SERVER_PROTOCOL",
23 "SERVER_SIGNATURE", "SERVER_SOFTWARE" };
24
25 cout << "Content-Type: text/html\n\n"; // gera saída do cabeçalho HTTP
26
27 // gera saída da declaração XML e DOCTYPE
28 cout << "<?xml version = \"1.0\"?>"
29 << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.1//EN\" \"
30 << \"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd\">";
31
32 // gera saída do elemento html e parte de seu conteúdo
33 cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
34 << "<head><title>Environment Variables</title></head><body>";
35
36 // começa a gerar saída de tabela
37 cout << "<table border = \"0\" cellspacing = \"2\">";
38
39 // itera por variáveis de ambiente
40 for (int i = 0; i < 24; i++)
41 {
42 cout << "<tr><td>" << environmentVariables[i] << "</td><td>";
43
44 // tenta recuperar o valor da variável de ambiente atual
45 char *value = getenv(environmentVariables[i].c_str());
46
47 if (value != 0) // variável de ambiente existe
48 cout << value;
49 else
50 cout << "Environment variable does not exist.";
51
52 cout << "</td></tr>";
53 } // fim do for
54
55 cout << "</table></body></html>";
56 return 0;
57 } // fim de main

```

**Figura 19.8** Recuperando variáveis de ambiente via função `getenv`.

(continua)



**Figura 19.8** Recuperando variáveis de ambiente via função getenv.

(continuação)

Na linha 16 da Figura 19.9, passamos "QUERY\_STRING" para a função `getenv`, que retorna a string de consulta ou um ponteiro nulo se o servidor não tiver configurado uma variável de ambiente `QUERY_STRING`. [Nota: O Apache HTTP Server configura `QUERY_STRING` mesmo se uma solicitação não contiver uma string de consulta — nesse caso, a variável contém uma string vazia. Entretanto, alguns servidores, como o IIS da Microsoft, configuram essa variável apenas se uma string de consulta realmente existir.] Se a variável de ambiente `QUERY_STRING` existir (isto é, `getenv` não retorna um ponteiro nulo), a linha 17 invoca `getenv` novamente, dessa vez atribuindo a string de consulta retornada à variável `string query`. Depois de gerar saída de um cabeçalho, alguns tags iniciais e o título XHTML (linhas 19–29), testamos se `query` contém dados (linha 32). Se não contiver, geramos saída de uma mensagem que instrui o usuário a adicionar uma string de consulta para o URL. Também fornecemos um link para um URL que inclui uma string de consulta de exemplo. Os dados de string de consulta podem ser especificados como parte de um hyperlink em uma página Web quando codificado dessa maneira. A saída do conteúdo da string de consulta é realizada pela linha 36.

Esse exemplo simplesmente demonstrou como acessar os dados passados para um script CGI na string de consulta. Os exemplos do capítulo posterior mostram como quebrar uma string de consulta em partes úteis de informações que podem ser manipuladas utilizando variáveis separadas.

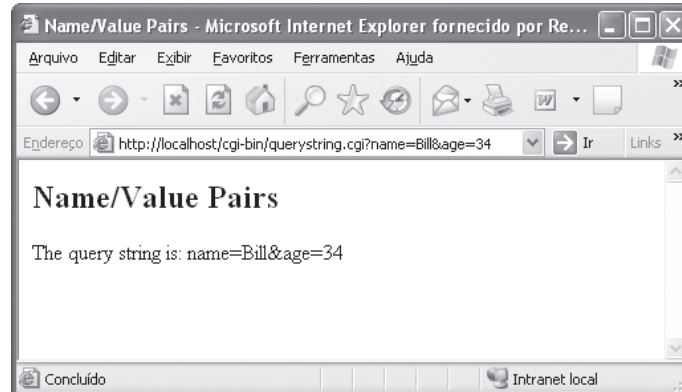
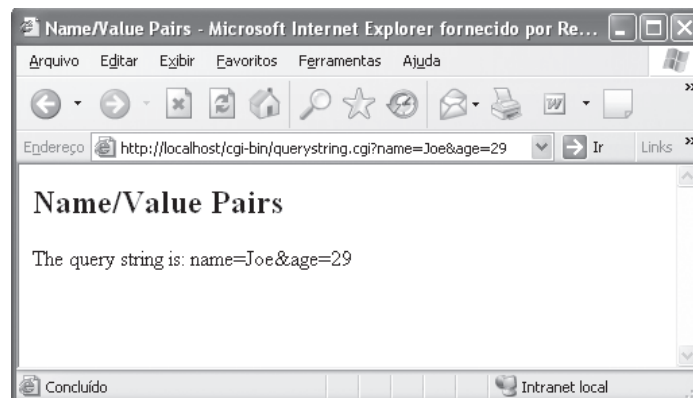
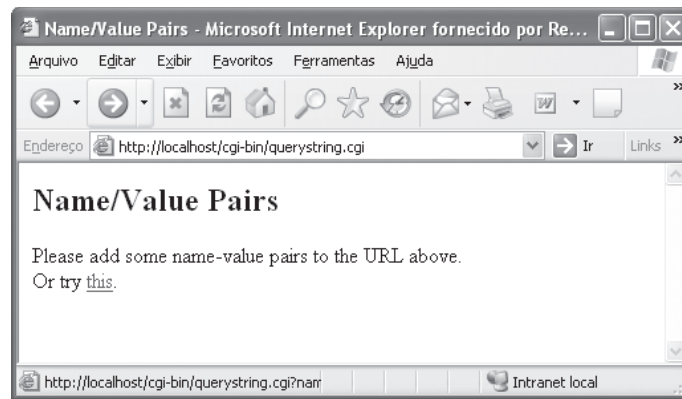
```

1 // Figura 19.9: querystring.cpp
2 // Demonstrando QUERY_STRING.
3 #include <iostream>
4 using std::cout;
5
6 #include <string>
7 using std::string;
8
9 #include <cstdlib>
10 using::getenv;
11
12 int main()
13 {
14 string query = "";
15
16 if (getenv("QUERY_STRING")) // A variável QUERY_STRING existe
17 query = getenv("QUERY_STRING"); // recupera o valor de QUERY_STRING
18
19 cout << "Content-Type: text/html\n\n"; // gera saída do cabeçalho HTTP
20
21 // gera saída da declaração XML e DOCTYPE
22 cout << "<?xml version = \"1.0\"?>"
23 << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.1//EN\" "
24 << "\"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd\">";
25
26 // gera saída do elemento html e parte de seu conteúdo
27 cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
28 << "<head><title>Name/Value Pairs</title></head><body>";
29 cout << "<h2>Name/Value Pairs</h2>";
30
31 // se a consulta não contiver dados
32 if (query == "")
33 cout << "Please add some name-value pairs to the URL above.
 Or"
34 << " try this.";
35 else // usuário inseriu string de consulta
36 cout << "<p>The query string is: " << query << "</p>";
37
38 cout << "</body></html>";
39 return 0;
40 } // fim de main

```

**Figura 19.9** Lendo a entrada a partir de `QUERY_STRING`.

(continua)



**Figura 19.9** Lendo a entrada a partir de QUERY\_STRING.

(continuação)

## 19.11 Utilizando formulários XHTML para enviar entrada

Fazer um cliente inserir a entrada diretamente em um URL não é uma abordagem amigável ao usuário. Felizmente, a XHTML fornece a capacidade de incluir formulários em páginas Web que fornecem aos usuários uma maneira mais intuitiva de inserir informações a serem enviadas para um script CGI.

### Elemento XHTML *form*

O elemento **form** inclui um formulário XHTML. O elemento **form** geralmente aceita dois atributos. O primeiro é **action**, que especifica o recurso de servidor a executar quando o usuário envia o formulário. Para nossos propósitos, o **action** normalmente será um script CGI que processa os dados do formulário. O segundo atributo utilizado no elemento **form** é **method**, que identifica o tipo de solicitação HTTP (isto é, **get** ou **post**) a utilizar quando o navegador envia o formulário para o servidor Web. Nesta seção, mostramos exemplos que utilizam tanto solicitações **get** como **post** para ilustrá-las em detalhes.

Um formulário XHTML pode conter qualquer quantidade de elementos. A Figura 19.10 fornece uma breve descrição de vários elementos de formulário.

### Utilizando a solicitação *get*

A Figura 19.11 demonstra um formulário XHTML básico utilizando o método HTTP *get*. A saída do formulário é gerada nas linhas 34–36 com o elemento `form`. Note que o atributo `method` tem o valor "get" e o atributo `action` tem o valor "getquery.cgi" (isto é, o script realmente chama a si próprio para tratar os dados de formulário, uma vez que ele é enviado).

O formulário contém dois campos `input`. O primeiro (linha 35) é um campo de texto de uma única linha (`type = "text"`) chamado `word`. O segundo (linha 36) exibe um botão, rotulado `Submit Word` (`value = "Submit Word"`), para enviar os dados de formulário.

Na primeira vez que o script for executado, não haverá nenhum valor em `QUERY_STRING` (a menos que o usuário tenha acrescentado a string de consulta ao URL). [Nota: Lembre-se de que em alguns servidores `QUERY_STRING` pode nem mesmo existir quando a string de consulta estiver vazia.] Uma vez que o usuário insere uma palavra no campo de texto `word` e clica em `Submit Word`, o script é novamente solicitado. Dessa vez, o nome do campo de entrada (`word`) e o valor inserido pelo usuário são colocados na variável de ambiente `QUERY_STRING`. Isto é, se o usuário insere a palavra 'technology' e clica em `Submit Word`, o valor `word=technology` é atribuído a `QUERY_STRING`. Observe que a string de consulta também é acrescentada ao URL no campo `Endereço` do navegador com um ponto de interrogação (?) na frente dela.

Durante a segunda execução do script, a string de consulta é decodificada. A linha 42 utiliza o método `string find_first_of` para pesquisar em `query` a primeira ocorrência de `word=`, que retorna um valor inteiro correspondente à localização na string onde a primeira correspondência foi localizada. A linha 42 então adiciona 5 ao valor retornado por `find_first_of` para configurar `wordLocation` como igual à posição da string contendo o primeiro caractere da palavra favorita do usuário. A função `substr` (linha 43) retorna o restante da string que inicia em `wordLocation`. A linha 43 então atribui essa string a `wordString`. A linha 45 determina se o usuário inseriu uma palavra. Em caso positivo, a linha 48 gera saída da palavra inserida pelo usuário.

Nome de elemento	Valor do atributo <code>type</code> (para elementos <code>input</code> )	Descrição
<b>input</b>	<code>text</code>	Fornece um campo de texto de uma única linha para a entrada de texto.
	<code>password</code>	Como <code>text</code> , mas cada caractere digitado pelo usuário aparece como um asterisco (*).
	<code>checkbox</code>	Exibe uma caixa de seleção que pode ser marcada ( <code>true</code> ) ou desmarcada ( <code>false</code> ).
	<code>radio</code>	Os botões de opção são como caixas de seleção, exceto pelo fato de que apenas um único botão de opção em um grupo de botões de opção pode ser selecionado por vez.
	<code>button</code>	Um botão push.
	<code>submit</code>	Um botão push que envia dados de formulário de acordo com o do formulário <code>action</code> .
	<code>image</code>	Igual a <code>submit</code> , mas exibe uma imagem em vez de um botão push.
	<code>reset</code>	Um botão push que redefine campos de formulário para seus valores-padrão.
	<code>file</code>	Exibe um campo de texto e um botão que permitem ao usuário especificar um arquivo para carregar em um servidor Web. Quando clicado, o botão abre uma caixa de diálogo de arquivos que permite ao usuário selecionar um arquivo.
	<code>hidden</code>	Dados de formulário ocultos que podem ser utilizados pelo handler de formulário no servidor. Esses <code>inputs</code> não são visíveis ao usuário.
<b>select</b>		Menu drop-down ou caixa de seleção. Este elemento é utilizado com o elemento <code>option</code> para especificar uma série de itens selecionáveis.
<b>textarea</b>		Este é um campo de texto de múltiplas linhas em que um texto pode ser inserido ou exibido.

**Figura 19.10** Elementos de formulário XHTML.

```

1 // Figura 19.11: getquery.cpp
2 // Demonstra o método GET com formulário XHTML.
3 #include <iostream>
4 using std::cout;
5
6 #include <string>
7 using std::string;
8
9 #include <cstdlib>
10 using std::getenv;
11
12 int main()
13 {
14 string nameString = "";
15 string wordString = "";
16 string query = "";
17
18 if (getenv("QUERY_STRING")) // A variável QUERY_STRING existe
19 query = getenv("QUERY_STRING"); // recupera o valor de QUERY_STRING
20
21 cout << "Content-Type: text/html\n\n"; // gera saída do cabeçalho HTTP
22
23 // gera saída da declaração XML e DOCTYPE
24 cout << "<?xml version = \"1.0\"?>"
25 << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.1//EN\" "
26 << "\"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd\">";
27
28 // gera saída do elemento html e parte de seu conteúdo
29 cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
30 << "<head><title>Using GET with Forms</title></head><body>";
31
32 // gera saída do formulário xhtml
33 cout << "<p>Enter one of your favorite words here:</p>"
34 << "<form method = \"get\" action = \"getquery.cgi\">"
35 << "<input type = \"text\" name = \"word\"/>"
36 << "<input type = \"submit\" value = \"Submit Word\"/></form>";
37
38 if (query == "") // a consulta está vazia
39 cout << "<p>Please enter a word.</p>";
40 else // usuário inseriu string de consulta
41 {
42 int wordLocation = query.find_first_of("word=") + 5;
43 wordString = query.substr(wordLocation);
44
45 if (wordString == "") // nenhuma palavra foi inserida
46 cout << "<p>Please enter a word.</p>";
47 else // uma palavra foi inserida
48 cout << "<p>Your word is: " << wordString << "</p>";
49 } // fim de else
50
51 cout << "</body></html>";
52 return 0;
53 } // fim de main

```

Figura 19.11 Utilizando o método get com um formulário XHTML.

(continua)



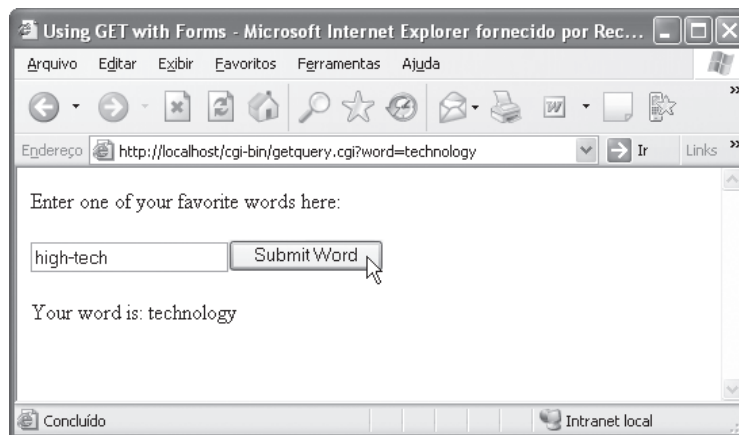
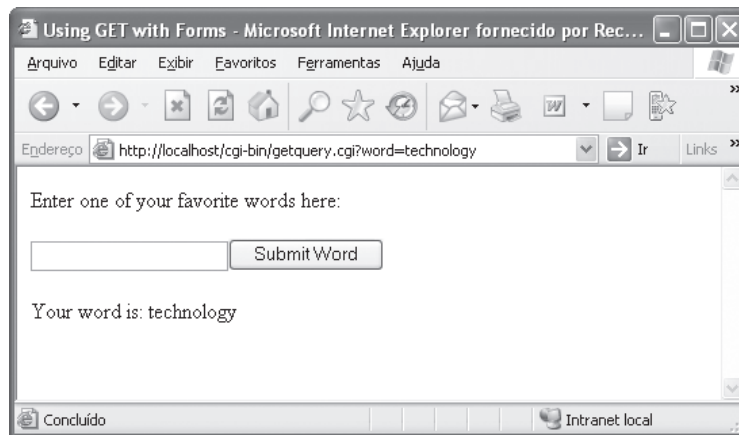
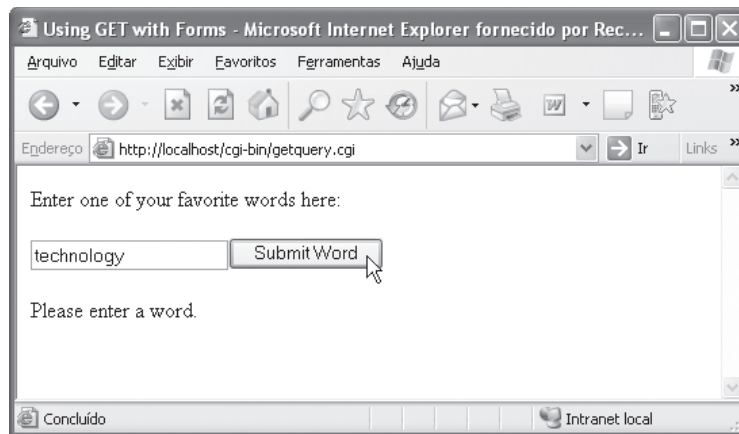


Figura 19.11 Utilizando o método get com um formulário XHTML.

(continua)

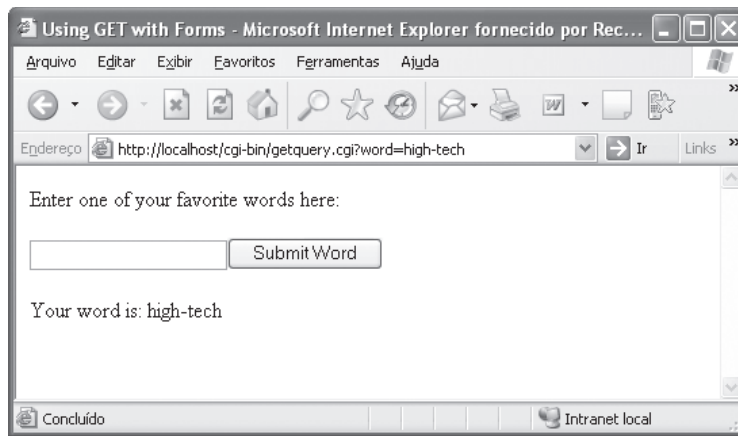


Figura 19.11 Utilizando o método get com um formulário XHTML.

(continuação)

### Utilizando a solicitação *post*

Os dois exemplos anteriores utilizaram *get* para passar dados para os scripts CGI por meio de uma variável de ambiente (isto é, `QUERY_STRING`). Em geral, os navegadores Web interagem com servidores Web enviando formulários utilizando *HTTP post*. Os programas CGI lêem o conteúdo de solicitações *post* utilizando entrada-padrão. Para propósitos de comparação, agora vamos reimplementar o aplicativo da Figura 19.11, utilizando o método *post* (como na Figura 19.12). Note que o código nas duas figuras é praticamente idêntico. O formulário XHTML (linhas 43–45) indica que agora estamos utilizando o método *post* para enviar os dados de formulário.

O servidor Web envia os dados *post* para um script CGI via entrada-padrão. Os dados são codificados (isto é, formatados) assim como em `QUERY_STRING` (isto é, com pares nome-valor conectados por um sinal de igual e um *e* comercial), mas a variável de ambiente `QUERY_STRING` não é configurada. Em vez disso, o método *post* configura a variável de ambiente `CONTENT_LENGTH`, para indicar quantos caracteres de dados foram enviados em uma solicitação *post*.

O script CGI utiliza o valor da variável de ambiente `CONTENT_LENGTH` para processar a quantidade correta de dados. A linha 23 determina se `CONTENT_LENGTH` contém um valor. Se contiver, a linha 25 lerá o valor e o converterá em um inteiro chamando a função `<cstdlib>` `atoi`. A linha 26 chama a função `cin.read` para ler caracteres a partir da entrada-padrão e os armazena no array `postString`. A linha 27 converte os dados de `postString` em uma string atribuindo-os a `dataString`.

Nos capítulos anteriores, lemos os dados de entrada-padrão utilizando uma expressão como

```
cin >> data;
```

A mesma abordagem poderia funcionar em nosso script CGI como uma substituição para a instrução `cin.read`. Lembre-se de que `cin` lê os dados a partir da entrada-padrão até e incluindo o primeiro caractere de nova linha, espaço ou tabulação, o que vier primeiro. A especificação CGI (livremente disponível em [cgi-spec.golux.com/cgi-120-00a.html](http://cgi-spec.golux.com/cgi-120-00a.html)) não requer que um caractere de nova linha seja acrescentado depois do último par nome-valor. Embora alguns navegadores acrescentem um caractere de nova linha ou `E0F`, eles não são necessários. Se `cin` for utilizado com um navegador que envia apenas os pares nome-valor (como ocorre pela especificação CGI), `cin` deve esperar por uma nova linha que nunca chegará. Nesse caso, o servidor irá, por fim, ‘expirar’ e o script CGI terminará. Portanto, `cin.read` é preferido a `cin`, porque o programador pode especificar exatamente quantos dados ler.

Os scripts CGI nesta seção, embora úteis para explicar como *get* e *post* operam, não incluem muitos dos recursos descritos na especificação CGI. Por exemplo, se inserirmos as palavras `didn't translate` no campo de texto e clicarmos no botão `submit`, o script nos informará que nossa palavra é `didn%27t+translate`.

O que aconteceu aqui? Os navegadores Web ‘URL codificam’ os dados de formulário XHTML que eles enviam. Isso quer dizer que os espaços são substituídos por sinais de adição, e outros símbolos (por exemplo, apóstrofes) são convertidos em seu valor ASCII no formato hexadecimal e precedidos por um sinal de porcentagem. A codificação URL é necessária porque os URLs não podem conter certos caracteres, como espaços e apóstrofes.

## 19.12 Outros cabeçalhos

Um script CGI pode fornecer outros cabeçalhos HTTP além de `Content-Type`. Na maioria dos casos, o servidor passa esses cabeçalhos extras para o cliente sem executá-los. Por exemplo, o cabeçalho **Refresh** a seguir redireciona o cliente para uma nova localização depois de uma quantidade especificada de tempo:

```
Refresh: "5; URL = http://www.deitel.com/newpage.html"
```

Cinco segundos depois de o navegador Web receber esse cabeçalho, o navegador solicita o recurso no URL especificado. Alternativamente, o cabeçalho **Refresh** pode omitir o URL, caso em que ele atualizará a página atual depois de o dado tempo ter expirado.

```

1 // Figura 19.12: post.cpp
2 // Demonstra o método POST com formulário XHTML.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6
7 #include <string>
8 using std::string;
9
10 #include <cstdlib>
11 using std::getenv;
12 using std::atoi;
13
14 int main()
15 {
16 char postString[1024] = ""; // variável para armazenar dados POST
17 string dataString = "";
18 string nameString = "";
19 string wordString = "";
20 int contentLength = 0;
21
22 // conteúdo foi enviado
23 if (getenv("CONTENT_LENGTH"))
24 {
25 contentLength = atoi(getenv("CONTENT_LENGTH"));
26 cin.read(postString, contentLength);
27 dataString = postString;
28 } // fim do if
29
30 cout << "Content-Type: text/html\n\n"; // gera saída do cabeçalho
31
32 // gera saída da declaração XML e DOCTYPE
33 cout << "<?xml version = \"1.0\"?>"
34 << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.1//EN\" "
35 << "\"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd\">";
36
37 // gera saída de elemento XHTML e parte de seu conteúdo
38 cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
39 << "<head><title>Using POST with Forms</title></head><body>";
40
41 // gera saída de formulário XHTML
42 cout << "<p>Enter one of your favorite words here:</p>"
43 << "<form method = \"post\" action = \"post.cgi\">"
44 << "<input type = \"text\" name = \"word\" />"
45 << "<input type = \"submit\" value = \"Submit Word\" /></form>";
46
47 // dados foram enviados utilizando POST
48 if (contentLength > 0)
49 {
50 int nameLocation = dataString.find_first_of("word=") + 5;
51 int endLocation = dataString.find_first_of("&") - 1;
52
53 // recupera palavra inserida
54 wordString = dataString.substr(
55 nameLocation, endLocation - nameLocation);
56
57 if (wordString == "") // nenhum dado foi inserido no campo de texto

```

Figura 19.12 Utilizando o método post com um formulário XHTML.

(continua)

```

58 cout << "<p>Please enter a word.</p>";
59 else // gera saída de palavra
60 cout << "<p>Your word is: " << wordString << "</p>";
61 } // fim do if
62 else // nenhum dado foi enviado
63 cout << "<p>Please enter a word.</p>";
64
65 cout << "</body></html>";
66 return 0;
67 } // fim de main

```

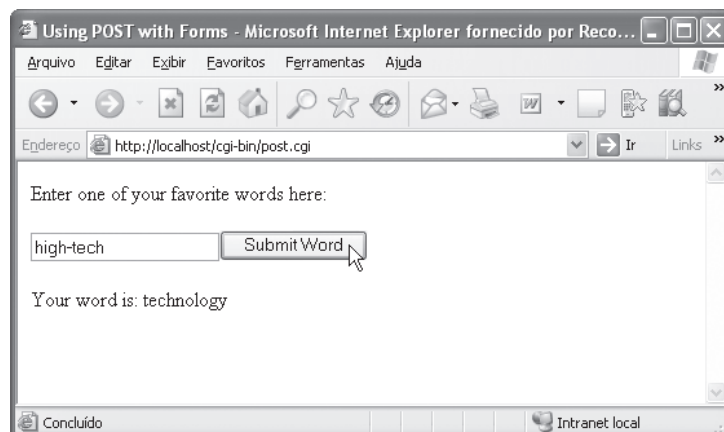
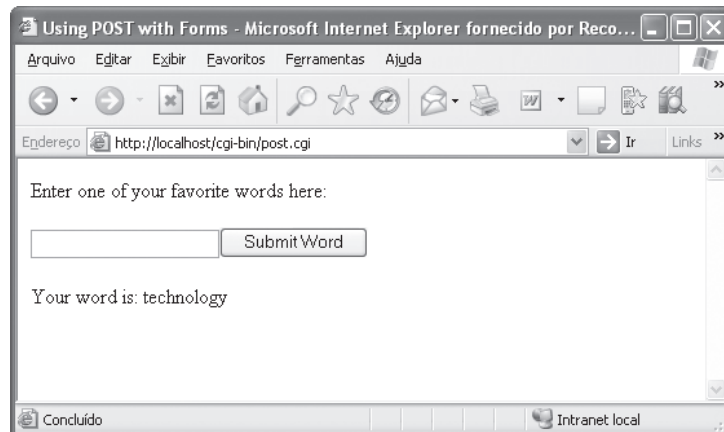
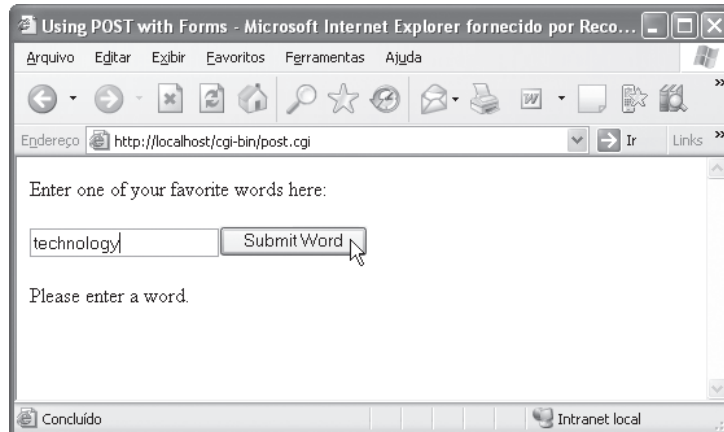
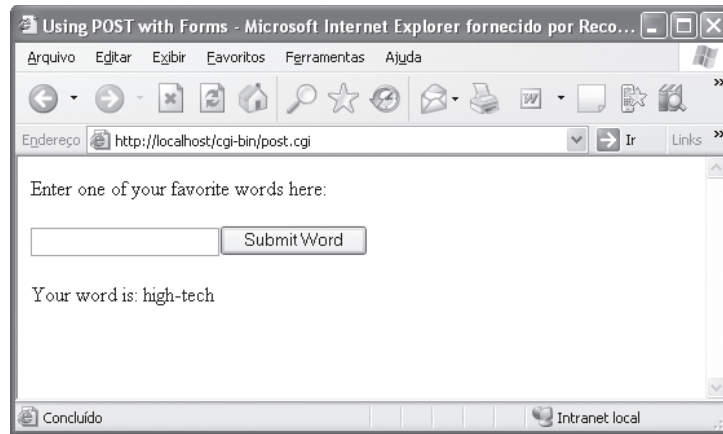


Figura 19.12 Utilizando o método post com um formulário XHTML.

(continua)



**Figura 19.12** Utilizando o método post com um formulário XHTML.

(continuação)

A especificação CGI indica que certos tipos de cabeçalhos cuja saída é realizada por um script CGI devem ser tratados pelo servidor, em vez de passados diretamente para o cliente. O primeiro desses é o **cabeçalho Location**. Como Refresh, Location redireciona o cliente para uma nova localização:

Location: `http://www.deitel.com/newpage.html`

Se utilizado com um URL relativo (ou virtual) (isto é, Location: `/newpage.html`), o cabeçalho Location indica ao servidor que o redirecionamento deve ser realizado no lado do servidor sem enviar o cabeçalho Location de volta para o cliente. Nesse caso, ele aparece para o usuário como se o documento renderizado no navegador Web fosse o recurso que o usuário havia solicitado, quando de fato o documento renderizado é o recurso especificado no cabeçalho Location.

A especificação CGI também inclui um **cabeçalho Status**, que instrui o servidor a gerar saída de uma linha de cabeçalho de status especificada (como HTTP/1.1 200 OK). Normalmente, o servidor enviará a linha de status apropriada para o cliente (adicionando, por exemplo, a linha de status 200 OK na maioria dos casos). Entretanto, o CGI permite aos programadores alterar o status de resposta. Por exemplo, enviar um cabeçalho

Status: 204 No Response

indica que, embora a solicitação seja bem-sucedida, o cliente não deve exibir uma nova página na janela de navegador. Esse cabeçalho poderia ser útil se você quisesse permitir aos usuários enviar formulários sem fazer uma relocação para uma nova página.

Agora já abordamos os fundamentos da especificação CGI. Para revisar, o CGI permite aos scripts interagir com servidores de três maneiras básicas:

1. pela saída de cabeçalhos e conteúdo para o servidor via saída-padrão;
2. pela configuração de variáveis de ambiente do servidor (incluindo a QUERY\_STRING codificada pelo URL) cujos valores estão disponíveis dentro do script (via getenv);
3. pelos dados codificados por URL e POSTados que o servidor envia à entrada-padrão do script.

### 19.13 Estudo de caso: uma página Web interativa

As figuras 19.13–19.14 mostram a implementação de um portal interativo simples para o site Web fictício da Bug2Bug Travel. O exemplo consulta o nome e a senha do cliente e, então, exibe informações sobre cada promoção de viagem semanal com base nos dados inseridos. Para simplificar, o exemplo não criptografa os dados enviados para o servidor. Idealmente, dados sensíveis, como uma senha, devem ser criptografados. A criptografia está além do escopo deste livro.

A Figura 19.13 exibe a página de abertura. Ela é um documento XHTML estático contendo um formulário que POSTa os dados para o script CGI `portal.cgi` (linha 16). O formulário contém um campo para coletar o nome do usuário (linha 18) e um para coletar a senha do usuário (linha 19). [Nota: Diferentemente dos scripts CGI, que são colocados no diretório `cgi-bin` do servidor Web, esse documento XHTML foi colocado no diretório `htdocs` do servidor Web.]

A Figura 19.14 contém o script CGI. Primeiro, vamos examinar como o nome e a senha do usuário são recuperados a partir da entrada-padrão e armazenados em strings. A função `find` da biblioteca `string` pesquisa uma ocorrência de `namebox=` em `dataString` (linha 30). A função `find` retorna uma localização na string em que `namebox=` foi encontrado. Para recuperar o valor associado com `namebox=` — o valor inserido pelo usuário — movemos a posição na string avançando 8 caracteres. O programa agora contém um inteiro que ‘aponta’ para a localização inicial. Lembre-se de que uma string de consulta contém pares nome-valor separados por sinal de igual

e comercial. Para encontrar a localização final dos dados que desejamos recuperar, procuramos o caractere & (linha 31). O comprimento da palavra inserida é determinado pelo cálculo `endNameLocation - nameLocation`. Utilizamos uma abordagem semelhante para determinar as localizações inicial e final da senha (linhas 32–33). As linhas 36–38 atribuem os valores de campo de formulário às variáveis `nameString` e `passwordString`. Utilizamos `nameString` na linha 52 para gerar saída de uma saudação personalizada para o usuário. Os pacotes semanais atuais são exibidos nas linhas 53–56.

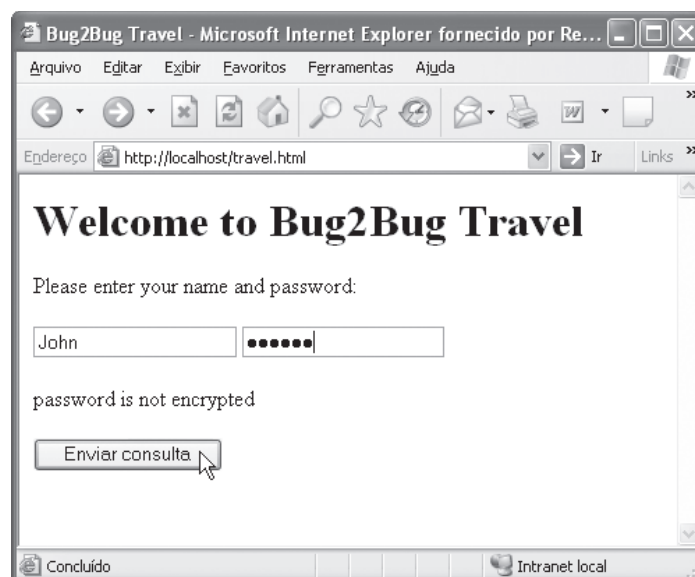
Se a senha-membro for correta, as linhas 59–60 geram saída de promoções adicionais. Se a senha for incorreta, o cliente é informado de que a senha é inválida e nenhuma promoção adicional é exibida.

Observe que aqui utilizamos uma página Web estática e um script CGI separado. Poderíamos ter incorporado o formulário XHTML de abertura e o processamento dos dados em um único script CGI, como fizemos nos exemplos anteriores deste capítulo. Solicitamos que o leitor faça isso no Exercício 19.8.

```

1 <?xml version = "1.0"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5 <!-- Figura 19.13: travel.html -->
6 <!-- Homepage da Bug2Bug Travel -->
7
8 <html xmlns = "http://www.w3.org/1999/xhtml">
9 <head>
10 <title>Bug2Bug Travel</title>
11 </head>
12
13 <body>
14 <h1>Welcome to Bug2Bug Travel</h1>
15
16 <form method = "post" action = "/cgi-bin/portal.cgi">
17 <p>Please enter your name and password:</p>
18 <input type = "text" name = "namebox" />
19 <input type = "password" name = "passwordbox" />
20 <p>password is not encrypted</p>
21 <input type = "submit" name = "button" />
22 </form>
23 </body>
24 </html>

```



**Figura 19.13** Portal interativo para criar uma página Web protegida por senha.

```

1 // Figura 19.14: portal.cpp
2 // Trata entrada em Bug2Bug Travel.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6
7 #include <string>
8 using std::string;
9
10 #include <cstdlib>
11 using std::getenv;
12 using std::atoi;
13
14 int main()
15 {
16 char postString[1024] = "";
17 string dataString = "";
18 string nameString = "";
19 string passwordString = "";
20 int contentLength = 0;
21
22 // dados foram postados
23 if (getenv("CONTENT_LENGTH"))
24 contentLength = atoi(getenv("CONTENT_LENGTH"));
25
26 cin.read(postString, contentLength);
27 dataString = postString;
28
29 // pesquisa dados de entrada na string
30 int namelocation = dataString.find("namebox=") + 8;
31 int endNamelocation = dataString.find("&");
32 int password = dataString.find("passwordbox=") + 12;
33 int endPassword = dataString.find("&button");
34
35 // obtém valores para nome e senha
36 nameString = dataString.substr(
37 namelocation, endNamelocation - namelocation);
38 passwordString = dataString.substr(password, endPassword - password);
39
40 cout << "Content-Type: text/html\n\n"; // gera saída do cabeçalho HTTP
41
42 // gera saída da declaração XML e DOCTYPE
43 cout << "<?xml version = \"1.0\"?>"
44 << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.1//EN\" "
45 << "\"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd\">";
46
47 // gera saída do elemento html e parte de seu conteúdo
48 cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
49 << "<head><title>Bug2Bug Travel</title></head><body>";
50
51 // gera saída das promoções
52 cout << "<h1>Welcome " << nameString << "!</h1>"
53 << "<p>Here are our weekly specials:</p>"
54 << "Boston to Taiwan ($875)"
55 << "San Diego to Hong Kong ($750)"
56 << "Chicago to Mexico City ($568)";

```

Figura 19.14 Handler de portal interativo.

(continua)



```

57
58 if (passwordString == "coast2coast") // a senha é correta
59 cout << "<hr /><p>Current member special: "
60 << "Seattle to Tokyo ($400)</p>";
61 else // senha era incorreta
62 cout << "<p>Sorry. You have entered an incorrect password</p>";
63
64 cout << "</body></html>";
65 return 0;
66 } // fim de main

```

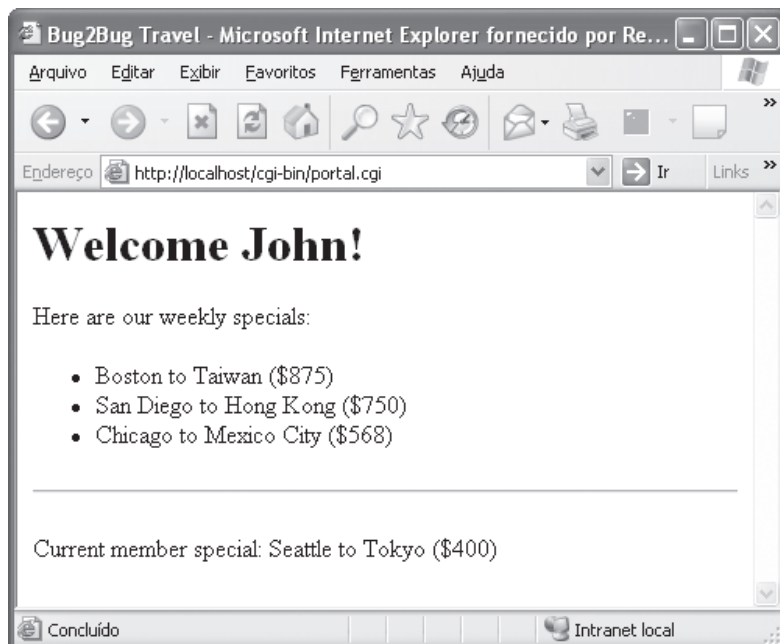
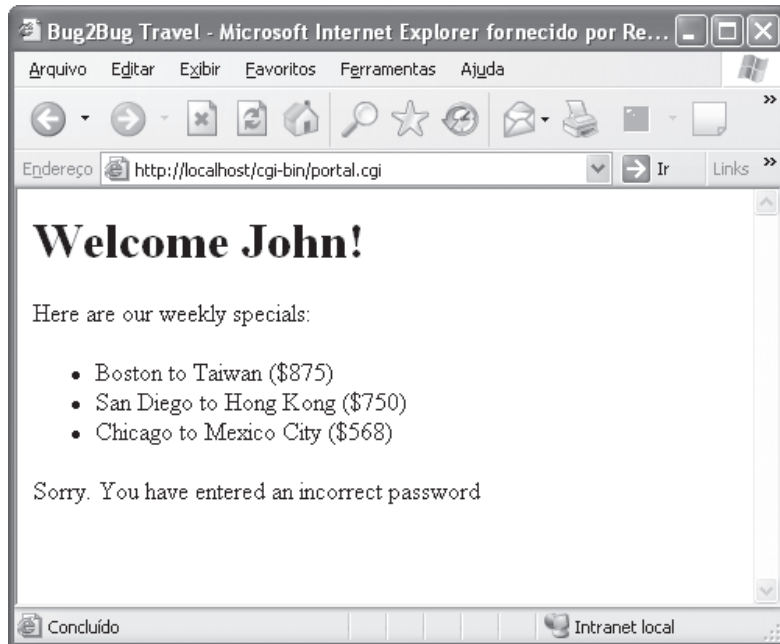


Figura 19.14 Handler de portal interativo.

(continuação)



## Dica de desempenho 19.1

*Para o servidor, é sempre muito mais eficiente fornecer conteúdo estático em vez de executar um script CGI, porque leva tempo para o servidor carregar o script do disco rígido para a memória e executá-lo (ao passo que um arquivo XHTML precisa ser enviado apenas para o cliente). É uma boa prática utilizar uma mistura de XHTML estática (para conteúdo que geralmente permanece inalterado) e script CGI (para conteúdo dinâmico). Essa prática permite ao servidor Web responder aos clientes com mais eficiência do que se o script CGI fosse utilizado.*

### 19.14 Cookies

Nas duas seções anteriores, discutimos duas maneiras de passar informações entre programas (ou execuções do mesmo programa) por meio de um navegador. Esta seção se concentra no armazenamento de informações de estado no computador-cliente com **cookies**. Cookies são essencialmente pequenos arquivos de texto que um servidor Web envia para o navegador, que então os salva no computador. Muitos sites Web utilizam cookies para monitorar o progresso dos usuários pelo site (como em um aplicativo de *shopping cart*) ou ajudar a personalizar o site para um usuário individual.

Os cookies não podem dominar seu computador, nem apagar a unidade de disco. Entretanto, podem ser utilizados para identificar usuários e monitorar a frequência com que eles visitam um site ou o que compram em um site. Por essa razão, os cookies são considerados uma questão de segurança e privacidade. Os navegadores Web populares fornecem suporte para cookies. Esses navegadores também permitem aos usuários preocupados com a privacidade e segurança desativar esse suporte. Os sites Web mais importantes utilizam cookies. Como um programador, você deve estar ciente da possibilidade de seus clientes desativarem os cookies. As figuras 19.15–19.17 utilizam cookies para armazenar e manipular informações sobre um usuário.

A Figura 19.15 é uma página XHTML que contém um formulário em que devem ser inseridos valores. O formulário posta suas informações para `writcookie.cgi` (Figura 19.16). Esse script CGI recupera os dados contidos na variável de ambiente `CONTENT_LENGTH`.

```

1 <?xml version = "1.0"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5 <!-- Figura 19.15: cookieform.html -->
6 <!-- Demonstração de cookie -->
7
8 <html xmlns = "http://www.w3.org/1999/xhtml">
9 <head>
10 <title>Writing a cookie to the client computer</title>
11 </head>
12
13 <body>
14 <h1>Click Submit to save your cookie data.</h1>
15
16 <form method = "post" action = "/cgi-bin/writcookie.cgi">
17 <p>Name:

18 <input type = "text" name = "name" />
19 </p>
20 <p>Age:

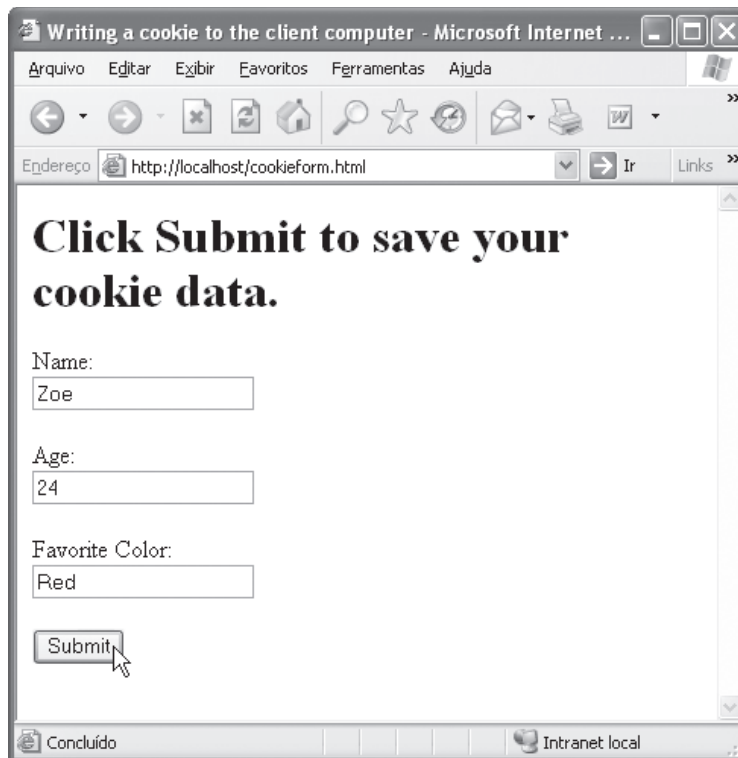
21 <input type = "text" name = "age" />
22 </p>
23 <p>Favorite Color:

24 <input type = "text" name = "color" />
25 </p>
26 <p>
27 <input type = "submit" name = "button" value = "Submit" />
28 </p>
29 </form>
30 </body>
31 </html>

```

**Figura 19.15** O documento XHTML contendo um formulário para postar dados para o servidor.

(continua)



**Figura 19.15** O documento XHTML contendo um formulário para postar dados para o servidor.

(continuação)

A linha 24 da Figura 19.16 declara e inicializa `string expires` para armazenar a data de expiração do cookie, a qual determina quanto tempo ele residirá na máquina do cliente. Esse valor pode ser uma string, como aquela neste exemplo, ou pode ser um valor relativo. Por exemplo, "+30d" configura o cookie para expirar depois de 30 dias. Para os propósitos deste capítulo, a data de expiração é deliberadamente configurada para expirar no ano 2010 a fim de assegurar que o programa executará de modo adequado futuramente. Você pode configurar a data de expiração desse exemplo para qualquer data futura como necessário. O navegador exclui cookies quando eles expiram.

Depois de obter os dados do formulário, o programa cria um cookie (linhas 54–56). Nesse exemplo, criamos um cookie que armazena uma linha de texto contendo o par nome-valor dos dados postados, delimitado por um caractere de dois-pontos (:). A saída da linha deve ser realizada antes de o cabeçalho ser gravado no cliente. A linha de texto inicia com o cabeçalho `Set-Cookie:`, que indica que o navegador deve armazenar os dados entrantes em um cookie. Configuramos três atributos para o cookie: um par nome-valor contendo o armazenamento de dados, um par nome-valor contendo a data de expiração e um par nome-valor contendo o URL do domínio de servidor (por exemplo, `www.deitel.com`) para os quais o cookie é válido. Para esse exemplo, `path` não é configurado como qualquer valor, tornando o cookie legível a partir de qualquer servidor no domínio do servidor que originalmente gravou o cookie. Observe que esses pares nome-valor são separados por ponto-e-vírgulas (;). Utilizamos somente caracteres de dois-pontos dentro de nossos dados de cookie para não haver conflito com o formato do cabeçalho `Set-Cookie:`. Quando inserimos os mesmos dados exibidos na Figura 19.15, as linhas 54–56 armazenam os dados "Name=Zoeage:24color:Red" no cookie. As linhas 59–76 enviam uma página Web que indica que o cookie foi gravado no cliente.



### Dica de portabilidade 19.1

*Os navegadores Web armazenam as informações do cookie de maneira específica do fornecedor. Por exemplo, o Microsoft Internet Explorer armazena cookies como arquivos de texto no diretório Arquivos temporários da Internet da máquina do cliente. O Netscape armazena seus cookies em um único arquivo chamado `cookies.txt`.*

A Figura 19.17 lê o cookie gravado na Figura 19.16 e exibe as informações armazenadas. Quando um cliente envia uma solicitação para um servidor, o navegador Web cliente localiza todos os cookies previamente gravados por esse servidor. Esses cookies são enviados pelo navegador de volta para o servidor como parte da solicitação. No servidor, a variável de ambiente `HTTP_COOKIE` armazena os cookies do cliente. A linha 20 chama a função `getenv` com a variável de ambiente `HTTP_COOKIE` como o parâmetro e armazena o valor retornado em `dataString`. Os pares nome-valor são decodificados e armazenados em strings (linhas 23–34) de acordo com o esquema de codificação utilizado na Figura 19.16. As linhas 36–55 geram saída do conteúdo do cookie como uma página Web.

```

1 // Figura 19.16: writecookie.cpp
2 // Programa para gravar um cookie na máquina de um cliente.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6
7 #include <string>
8 using std::string;
9
10 #include <cstdlib>
11 using std::getenv;
12 using std::atoi;
13
14 int main()
15 {
16 char query[1024] = "";
17 string dataString = "";
18 string nameString = "";
19 string ageString = "";
20 string colorString = "";
21 int contentLength = 0;
22
23 // data de expiração do cookie
24 string expires = "Friday, 14-MAY-10 16:00:00 GMT";
25
26 // dados foram inseridos
27 if (getenv("CONTENT_LENGTH"))
28 {
29 contentLength = atoi(getenv("CONTENT_LENGTH"));
30 cin.read(query, contentLength); // lê dados a partir da entrada-padrão
31 dataString = query;
32
33 // pesquisa por dados na string e armazena localizações
34 int nameLocation = dataString.find("name=") + 5;
35 int endName = dataString.find("&");
36 int ageLocation = dataString.find("age=") + 4;
37 int endAge = dataString.find("&color");
38 int colorLocation = dataString.find("color=") + 6;
39 int endColor = dataString.find("&button");
40
41 // obtém o valor do nome do usuário
42 nameString = dataString.substr(
43 nameLocation, endName - nameLocation);
44
45 if (ageLocation > 0) // obtém o valor da idade do usuário
46 ageString = dataString.substr(
47 ageLocation, endAge - ageLocation);
48
49 if (colorLocation > 0) // obtém o valor da cor favorita do usuário
50 colorString = dataString.substr(
51 colorLocation, endColor - colorLocation);
52
53 // configura o cookie
54 cout << "Set-Cookie: Name=" << nameString << "age:"
55 << ageString << "color:" << colorString
56 << "; expires=" << expires << "; path=\n";

```

Figura 19.16 Gravando um cookie.

(continua)

```

57 } // fim do if
58
59 cout << "Content-Type: text/html\n\n"; // gera saída do cabeçalho HTTP
60
61 // gera saída da declaração XML e DOCTYPE
62 cout << "<?xml version = \"1.0\"?>"
63 << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.1//EN\" "
64 << "\"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd\">";
65
66 // gera saída do elemento html e parte de seu conteúdo
67 cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
68 << "<head><title>Cookie Saved</title></head><body>";
69
70 // gera saída de informações do usuário
71 cout << "<p>A cookie has been set with the following"
72 << " data:</p><p>Name: " << nameString << "
</p>"
73 << "<p>Age: " << ageString << "
</p>"
74 << "<p>Color: " << colorString << "
</p>"
75 << "<p>Click "
76 << "here to read saved cookie data.</p></body></html>";
77 return 0;
78 } // fim de main

```

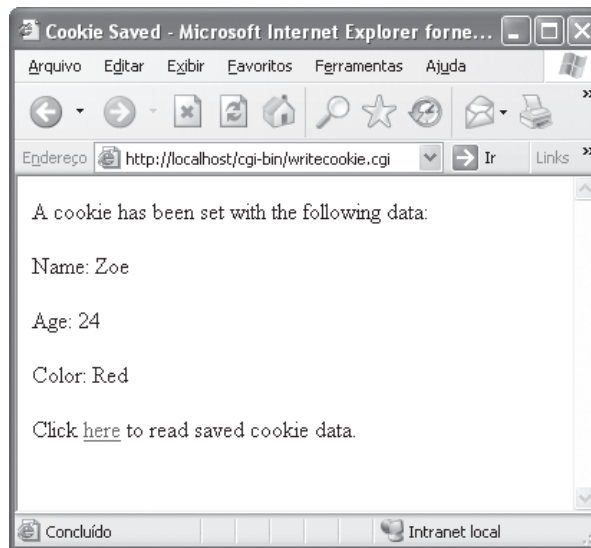


Figura 19.16 Gravando um cookie.

(continuação)



## Observação de engenharia de software 19.2

Os cookies apresentam um risco de segurança. Se usuários não autorizados ganham acesso a um computador, eles podem examinar o disco local e visualizar arquivos, que incluem cookies. Por essa razão, os dados sensíveis, como senhas, CIC e números de cartão de crédito, nunca devem ser armazenados em cookies.

## 19.15 Arquivos do lado do servidor

Na seção anterior, demonstramos como manter informações de estado sobre um usuário via cookies. O outro mecanismo para fazer isso é criar **arquivos do lado do servidor** (isto é, arquivos que são localizados no servidor ou na rede do servidor). Esse é um método de manutenção de informações vitais ligeiramente mais seguro. Nesse mecanismo, somente alguém com acesso e permissão para mudar arquivos no servidor pode alterar arquivos. As figuras 19.18–19.19 solicitam informações de contato aos usuários, e então as armazenam no servidor. A Figura 19.20 mostra o arquivo que é criado pelo script.

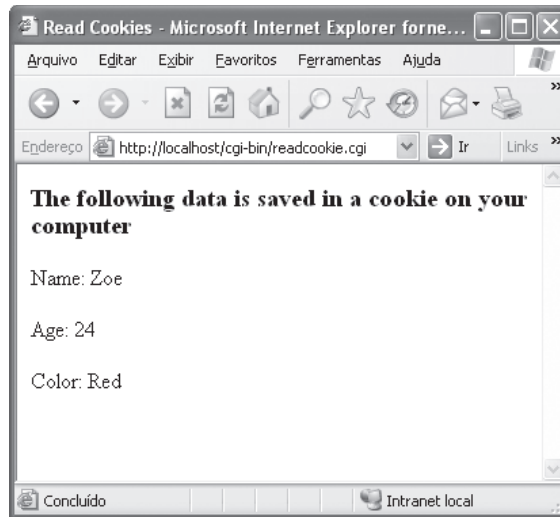
```

1 // Figura 19.17: readcookie.cpp
2 // Programa para ler dados de cookie.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6
7 #include <string>
8 using std::string;
9
10 #include <cstdlib>
11 using std::getenv;
12
13 int main()
14 {
15 string dataString = "";
16 string nameString = "";
17 string ageString = "";
18 string colorString = "";
19
20 dataString = getenv("HTTP_COOKIE"); // obtém dados de cookie
21
22 // pesquisa dados de cookie na string
23 int nameLocation = dataString.find("Name=") + 5;
24 int endName = dataString.find("age:");
25 int ageLocation = dataString.find("age:") + 4;
26 int endAge = dataString.find("color:");
27 int colorLocation = dataString.find("color:") + 6;
28
29 // armazena dados de cookie em strings
30 nameString = dataString.substr(
31 nameLocation, endName - nameLocation);
32 ageString = dataString.substr(
33 ageLocation, endAge - ageLocation);
34 colorString = dataString.substr(colorLocation);
35
36 cout << "Content-Type: text/html\n\n"; // gera saída do cabeçalho HTTP
37
38 // gera saída da declaração XML e DOCTYPE
39 cout << "<?xml version = \"1.0\"?>"
40 << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.1//EN\" "
41 << "\"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd\">";
42
43 // gera saída do elemento html e parte de seu conteúdo
44 cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
45 << "<head><title>Read Cookies</title></head><body>";
46
47 if (dataString != "") // dados foram localizados
48 cout << "<h3>The following data is saved in a cookie on"
49 << " your computer</h3><p>Name: " << nameString << "
</p>"
50 << "<p>Age: " << ageString << "
</p>"
51 << "<p>Color: " << colorString << "
</p>";
52 else // nenhum dado foi localizado
53 cout << "<p>No cookie data.</p>";
54
55 cout << "</body></html>";
56 return 0;
57 } // fim de main

```

**Figura 19.17** Programa para ler cookies enviados a partir do computador do cliente.

(continua)



**Figura 19.17** Programa para ler cookies enviados a partir do computador do cliente.

(continuação)

O documento XHTML da Figura 19.18 posta os dados de formulário no script CGI da Figura 19.19. No script CGI, as linhas 45–92 decodificam os parâmetros que foram enviados pelo cliente. A linha 105 cria uma instância do fluxo de arquivo de saída (outFile) que abre um arquivo para acréscimo de dados. Se o arquivo `clients.txt` não existir, ele será criado. As linhas 114–116 geram saída das informações pessoais para o arquivo. (Consulte a Figura 19.20 para ver o conteúdo do arquivo.) O restante do programa gera saída de um documento XHTML que resume as informações do usuário.

```

1 <?xml version = "1.0"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5 <!-- Figura 19.18: savefile.html -->
6 <!-- Formulário para inserir informações do cliente -->
7
8 <html xmlns = "http://www.w3.org/1999/xhtml">
9 <head>
10 <title>Please enter your contact information</title>
11 </head>
12
13 <body>
14 <p>Please enter your information in the form below.</p>
15 <p>Note: You must fill in all fields.</p>
16 <form method = "post" action = "/cgi-bin/savefile.cgi">
17 <p>
18 First Name:
19 <input type = "text" name = "firstname" size = "10" />
20 Last Name:
21 <input type = "text" name = "lastname" size = "15" />
22 </p>
23 <p>
24 Address:
25 <input type = "text" name = "address" size = "25" />

26 Town: <input type = "text" name = "town" size = "10" />
27 State: <input type = "text" name = "state" size = "2" />


```

**Figura 19.18** Documento XHTML para ler informações de contato do usuário.

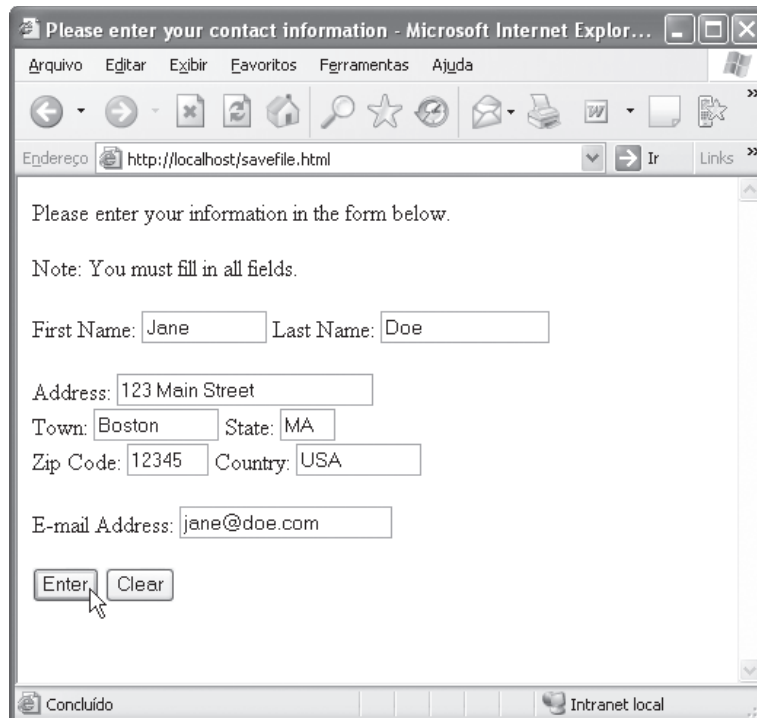
(continua)



```

28 Zip Code: <input type = "text" name = "zipcode" size = "5" />
29 Country: <input type = "text" name = "country" size = "10" />
30 </p>
31 <p>
32 E-mail Address: <input type = "text" name = "email" />
33 </p>
34 <input type = "submit" value = "Enter" />
35 <input type = "reset" value = "Clear" />
36 </form>
37 </body>
38 </html>

```



**Figura 19.18** Documento XHTML para ler informações de contato do usuário.

(continuação)

```

1 // Figura 19.19: savefile.cpp
2 // Programa para inserir informações de contato do cliente em um
3 // arquivo do lado do servidor.
4 #include <iostream>
5 using std::cerr;
6 using std::cin;
7 using std::cout;
8 using std::ios;
9
10 #include <fstream>
11 using std::ofstream;
12
13 #include <string>
14 using std::string;
15

```

**Figura 19.19** Criando um arquivo do lado do servidor para armazenar dados do usuário.

(continua)

```

16 #include <cstdlib>
17 using std::getenv;
18 using std::atoi;
19 using std::exit;
20
21 int main()
22 {
23 char postString[1024] = "";
24 int contentLength = 0;
25
26 // variáveis para armazenar dados do usuário
27 string dataString = "";
28 string firstname = "";
29 string lastname = "";
30 string address = "";
31 string town = "";
32 string state = "";
33 string zipcode = "";
34 string country = "";
35 string email = "";
36
37 // dados foram postados
38 if (getenv("CONTENT_LENGTH"))
39 contentLength = atoi(getenv("CONTENT_LENGTH"));
40
41 cin.read(postString, contentLength);
42 dataString = postString;
43
44 // procura o primeiro caractere '+'
45 string::size_type charLocation = dataString.find("+");
46
47 // procura o próximo caractere '+'
48 while (charLocation < string::npos)
49 {
50 dataString.replace(charLocation, 1, " ");
51 charLocation = dataString.find("+", charLocation + 1);
52 } // fim do while
53
54 // encontra a localização do nome
55 int firstStart = dataString.find("firstname=") + 10;
56 int endFirst = dataString.find("&lastname");
57 firstname = dataString.substr(firstStart, endFirst - firstStart);
58
59 // encontra a localização do sobrenome
60 int lastStart = dataString.find("lastname=") + 9;
61 int endLast = dataString.find("&address");
62 lastname = dataString.substr(lastStart, endLast - lastStart);
63
64 // encontra a localização do endereço
65 int addressStart = dataString.find("address=") + 8;
66 int endAddress = dataString.find("&town");
67 address = dataString.substr(addressStart, endAddress - addressStart);
68
69 // encontra a localização da cidade
70 int townStart = dataString.find("town=") + 5;
71 int endTown = dataString.find("&state");
72 town = dataString.substr(townStart, endTown - townStart);

```

**Figura 19.19** Criando um arquivo do lado do servidor para armazenar dados do usuário.

(continua)

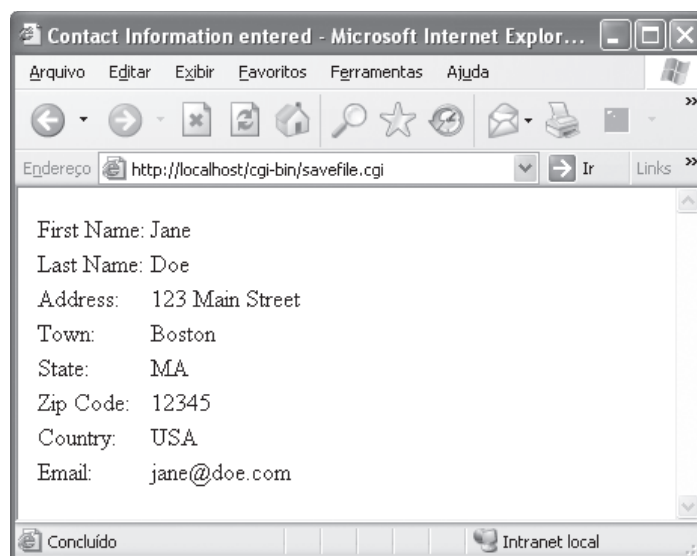
```

73
74 // encontra a localização do estado
75 int stateStart = dataString.find("state=") + 6;
76 int endState = dataString.find("&zipcode");
77 state = dataString.substr(stateStart, endState - stateStart);
78
79 // encontra a localização do CEP
80 int zipStart = dataString.find("zipcode=") + 8;
81 int endZip = dataString.find("&country");
82 zipcode = dataString.substr(zipStart, endZip - zipStart);
83
84 // encontra a localização do país
85 int countryStart = dataString.find("country=") + 8;
86 int endCountry = dataString.find("&email");
87 country = dataString.substr(countryStart, endCountry - countryStart);
88
89 // encontra a localização do e-mail
90 int emailStart = dataString.find("email=") + 6;
91 int endEmail = dataString.find("&submit");
92 email = dataString.substr(emailStart, endEmail - emailStart);
93
94 cout << "Content-Type: text/html\n\n"; // gera saída do cabeçalho
95
96 // gera saída da declaração XML e DOCTYPE
97 cout << "<?xml version = \"1.0\"?>"
98 << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.1//EN\" "
99 << "\"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd\">";
100
101 // gera saída do elemento html e parte de seu conteúdo
102 cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
103 << "<head><title>Contact Information entered</title></head><body>";
104
105 ofstream outFile("clients.txt", ios::app); // gera saída para o arquivo
106
107 if (!outFile) // arquivo não foi aberto adequadamente
108 {
109 cerr << "Error: could not open contact file.";
110 exit(1);
111 } // fim do if
112
113 // acrescenta dados ao arquivo clients.txt
114 outFile << firstname << " " << lastname << "\n" << address << "\n"
115 << town << " " << state << " " << country << " " << zipcode
116 << "\n" << email << "\n\n";
117
118 // gera saída de dados para o usuário
119 cout << "<table><tbody><tr><td>First Name:</td><td>" << firstname
120 << "</td></tr><tr><td>Last Name:</td><td>" << lastname
121 << "</td></tr><tr><td>Address:</td><td>" << address
122 << "</td></tr><tr><td>Town:</td><td>" << town
123 << "</td></tr><tr><td>State:</td><td>" << state
124 << "</td></tr><tr><td>Zip Code:</td><td>" << zipcode
125 << "</td></tr><tr><td>Country:</td><td>" << country
126 << "</td></tr><tr><td>Email:</td><td>" << email
127 << "</td></tr></tbody></table></body>\n</html>\n";
128 return 0;
129 } // fim de main

```

**Figura 19.19** Criando um arquivo do lado do servidor para armazenar dados do usuário.

(continua)



**Figura 19.19** Criando um arquivo do lado do servidor para armazenar dados do usuário.

(continuação)

```
Jane Doe
123 Main Street
Boston MA USA 12345
jane@doe.com
```

**Figura 19.20** Conteúdo do arquivo de dados clients.txt.

Há alguns pontos importantes a destacar sobre esse programa. Primeiro, não realizamos nenhuma validação nos dados antes de gravá-los no disco. Normalmente, o script verificaria os dados ruins, incompletos etc. Segundo, nosso arquivo está localizado no diretório cgi-bin, que é publicamente acessível. Alguém que conhecesse o nome de arquivo acharia relativamente fácil acessar informações de contato de outra pessoa.

Esse script não é robusto o bastante para implantação na Internet, mas fornece um exemplo do uso de arquivos do lado do servidor para armazenar informações. Uma vez que os arquivos são armazenados no servidor, os usuários não podem alterá-los a menos que tenham permissão do administrador de servidor para fazer isso. Portanto, armazenar esses arquivos no servidor é mais seguro que armazenar dados de usuários em cookies. [Nota: Muitos sistemas armazenam informações de usuário em bancos de dados protegidos por senha para obter níveis mais altos de segurança.]

Observe que, neste exemplo, mostramos como gravar dados em um arquivo do lado do servidor. Na próxima seção mostraremos como recuperar os dados a partir de um arquivo do lado do servidor, utilizando as técnicas utilizadas para ler de um arquivo apresentadas no Capítulo 17, “Processamento de arquivo”.

## 19.16 Estudo de caso: carrinho de compras

Sites Web de muitas empresas contêm aplicativos de carrinho de compras, que permitem aos clientes comprar itens de modo conveniente. Os sites registram o que o consumidor quer comprar e fornecem uma maneira intuitiva e fácil de fazer compras on-line. Isso é feito usando um carrinho de compras eletrônico, de modo semelhante ao uso de carrinhos de supermercados. À medida que usuários adicionam itens aos carrinhos de compras, os sites atualizam o conteúdo dos carinhos. Ao fazer o ‘check out’, os usuários pagam os itens incluídos nos carrinhos de compras. Para ver um carrinho de compras eletrônico do mundo real, sugerimos uma visita à livraria on-line Amazon.com (www.amazon.com).

O carrinho de compras implementado nesta seção (figuras 19.21–19.24) permite aos usuários comprar livros de uma livraria fictícia que vende quatro livros (ver Figura 19.23). Esse exemplo utiliza quatro scripts, dois arquivos do lado do servidor e dois cookies.

A Figura 19.21 mostra o primeiro desses scripts — a página de login. Esse script é o mais complexo de todos os scripts desta seção. A primeira condição if (linha 37) determina se os dados foram postados para o programa. A segunda condição if (linha 66) determina se dataString permanece vazia (isto é, não há dados enviados para decodificação, ou a decodificação não foi completada com sucesso). Na primeira vez que executarmos esse programa, a primeira condição irá falhar e a segunda condição será bem-sucedida. Desse modo

as linhas 72–77 irão gerar saída de um formulário XHTML para o usuário, como mostrado na primeira captura de tela da Figura 19.21. Quando o usuário preenche o formulário e clica no botão login, `login.cgi` é solicitado novamente — dessa vez, a solicitação contém dados postados, então a condição da linha 37 é avaliada como `true` e a da linha 66 é avaliada como `false`.

Se o usuário enviou os dados, o controle do programa continua no bloco `else` que inicia na linha 79, onde o script processa os dados. A linha 86 abre `userdata.txt` — o arquivo que contém todos os nomes de usuário e senhas para membros existentes. Se o usuário marcou a caixa de seleção New na página Web para criar um novo membro, a condição na linha 95 é avaliada como `true` e o script tenta registrar as informações do usuário no arquivo `userdata.txt` do servidor. As linhas 98–102 lêem por esse arquivo, comparando cada nome de usuário com o nome inserido. Se o nome de usuário já aparece no arquivo, o loop das linhas 98–102 termina antes de alcançar o fim do arquivo, as linhas 107–108 geram saída de uma mensagem apropriada para o usuário, e um hyperlink para voltar ao formulário é fornecido. Se o nome de usuário inserido ainda não existe em `userdata.txt`, a linha 117 adiciona as novas informações de usuário ao arquivo no formato

```
Bernard
blue
```

Cada nome de usuário e senha é separado por um caractere de nova linha. As linhas 119–120 fornecem um hyperlink para o script da Figura 19.22, que permite aos usuários comprar itens.

O último possível cenário para esse script é o retorno de usuários (linhas 123–162). Essa parte do programa executa quando o usuário insere um nome e uma senha, mas não seleciona a caixa de seleção New (isto é, o `else` de linha 123 é avaliado). Nesse caso, supomos que o usuário já tem um nome de usuário e uma senha em `userdata.txt`. As linhas 128–139 lêem por `userdata.txt` em uma tentativa de localizar o nome de usuário inserido. Se o nome de usuário for localizado (linha 131), determinamos se a senha inserida corresponde ou não àquela armazenada no arquivo (linha 137). Se corresponder, a variável `bool authenticated` é configurada como `true`. Caso contrário, `authenticated` permanece `false`. Se o usuário tiver sido autenticado (linha 142), a linha 144 chama a função `writeCookie` para inicializar um cookie chamado CART (linha 188), que é utilizado por outros scripts para armazenar dados que indicam os livros que o usuário adicionou ao carrinho de compras. Observe que esse cookie substitui qualquer cookie existente do mesmo nome, fazendo com que os dados de sessões anteriores sejam excluídos. Depois de criar o cookie, o script gera saída de uma mensagem que recebe o usuário de volta para o site Web e que fornece um link para `shop.cgi`, onde o usuário pode comprar livros (linhas 147–148).

Se o usuário não foi autenticado, o programa determina a razão (linhas 154–160). Se o usuário é localizado mas não autenticado, uma mensagem é gerada para indicar que a senha é inválida (linha 155–157). Um hyperlink é fornecido para a página de login (`<a href="/cgi-bin/login.cgi">`), onde o usuário pode tentar efetuar login novamente. Se nenhum nome de usuário ou senha é localizado, um usuário não registrado tentou efetuar login. As linhas 159–160 geram saída de uma mensagem que indica que o usuário não tem a autorização adequada para acessar a página e que fornece um link que permite ao usuário tentar outro login.

```
1 // Figura 19.21: login.cpp
2 // Programa para gerar saída de um formulário XHTML, verificar o
3 // nome do usuário e a senha inseridos e adicionar membros.
4 #include <iostream>
5 using std::cerr;
6 using std::cin;
7 using std::cout;
8 using std::ios;
9
10 #include <fstream>
11 using std::fstream;
12
13 #include <string>
14 using std::string;
15
16 #include <cstdlib>
17 using std::getenv;
18 using std::atoi;
19 using std::exit;
20
21 void header();
22 void writeCookie();
23
```

**Figura 19.21** Programa que gera saída de uma página de login.

(continua)

```

24 int main()
25 {
26 char query[1024] = "";
27 string dataString = "";
28
29 // strings para armazenar nome de usuário e senha
30 string userName = "";
31 string passWord = "";
32
33 int contentLength = 0;
34 bool newMember = false;
35
36 // dados foram postados
37 if (getenv("CONTENT_LENGTH"))
38 {
39 // recupera string de consulta
40 contentLength = atoi(getenv("CONTENT_LENGTH"));
41 cin.read(query, contentLength);
42 dataString = query;
43
44 // encontra a localização do nome de usuário
45 int userLocation = dataString.find("user=") + 5;
46 int endUser = dataString.find("&");
47
48 // encontra a localização da senha
49 int passwordLocation = dataString.find("password=") + 9;
50 int endPassword = dataString.find("&new");
51
52 if (endPassword > 0) // exige que o usuário se torne um novo membro
53 {
54 newMember = true;
55 passWord = dataString.substr(
56 passwordLocation, endPassword - passwordLocation);
57 } // fim do if
58 else // membro existente
59 passWord = dataString.substr(passwordLocation);
60
61 userName = dataString.substr(
62 userLocation, endUser - userLocation);
63 } // fim do if
64
65 // nenhum dado foi recuperado
66 if (dataString == "")
67 {
68 header();
69 cout << "<p>Please login.</p>";
70
71 // gera saída de formulário de login
72 cout << "<form method = \"post\" action = \"/cgi-bin/login.cgi\">"
73 << "<p>User Name: <input type = \"text\" name = \"user\"/>
"
74 << "Password: <input type = \"password\" name = \"password\"/>"
75 << "
New? <input type = \"checkbox\" name = \"new\" "
76 << " value = \"1\"/></p>"
77 << "<input type = \"submit\" value = \"login\"/></form>";
78 } // fim do if
79 else // processo inseriu dados

```

**Figura 19.21** Programa que gera saída de uma página de login.

(continua)

```

80 {
81 string fileUsername = "";
82 string filePassword = "";
83 bool userFound = false;
84
85 // abre arquivo de dados de usuário para leitura e gravação
86 fstream userData("userdata.txt", ios::in | ios::out);
87
88 if (!userData) // não foi possível abrir arquivo
89 {
90 cerr << "Could not open database.";
91 exit(1);
92 } // fim do if
93
94 // adiciona novo membro
95 if (newMember)
96 {
97 // lê nome de usuário e senha a partir do arquivo
98 while (!userFound && userData >> fileUsername >> filePassword)
99 {
100 if (userName == fileUsername) // nome já foi aceito
101 userFound = true;
102 } // fim do while
103
104 if (userFound) // nome de usuário foi aceito
105 {
106 header();
107 cout << "<p>This name has already been taken.</p>"
108 << "Try Again";
109 } // fim do if
110 else // processa os dados
111 {
112 writeCookie(); // grava o cookie
113 header();
114
115 // grava dados de usuário no arquivo
116 userData.clear(); // limpa eof, permite gravar no fim do arquivo
117 userData << "\n" << userName << "\n" << passWord;
118
119 cout << "<p>Your information has been processed."
120 << "Start Shopping</p>";
121 } // fim de else
122 } // fim do if
123 else // procura senha se inserida
124 {
125 bool authenticated = false;
126
127 // lê dados de usuário
128 while (!userFound && userData >> fileUsername >> filePassword)
129 {
130 // nome de usuário foi localizado
131 if (userName == fileUsername)
132 {
133 userFound = true;
134

```

**Figura 19.21** Programa que gera saída de uma página de login.

(continua)



```

135 // determina se a senha é correta
136 // e atribui resultado bool a authenticated
137 authenticated = (passWord == filePassword);
138 } // fim do if
139 } // fim do while
140
141 // o usuário é autenticado
142 if (authenticated)
143 {
144 writeCookie();
145 header();
146
147 cout << "<p>Thank you for returning, " << userName << "!</p>"
148 << "Start Shopping";
149 } // fim do if
150 else // usuário não autenticado
151 {
152 header();
153
154 if (userFound) // senha é incorreta
155 {
156 cout << "<p>You have entered an incorrect password. "
157 << "Please try again.</p>"
158 << "Back to login";
159 }
160 else // o usuário não é registrado
161 {
162 cout << "<p>You are not a registered user.</p>"
163 << "Register";
164 } // fim do else
165 } // fim do else
166 } // fim do else
167 } // fim do else
168
169 cout << "</body>\n</html>\n";
170 return 0;
171 } // fim do main
172
173 // função para gerar saída do cabeçalho
174 void header()
175 {
176 cout << "Content-Type: text/html\n\n"; // gera saída do cabeçalho
177
178 // gera saída da declaração XML e DOCTYPE
179 cout << "<?xml version = \"1.0\"?>"
180 << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.1//EN\" "
181 << "\"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd\">";
182
183 // gera saída do elemento html e parte de seu conteúdo
184 cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
185 << "<head><title>Login Page</title></head><body>";
186 } // fim da função header
187
188 // função para gravar dados de cookie
189 void writeCookie()
190 {
191 string expires = "Friday, 14-MAY-10 16:00:00 GMT";
192 cout << "Set-Cookie: CART=; expires=" << expires << "; path=\n";
193 } // fim da função writeCookie

```

**Figura 19.21** Programa que gera saída de uma página de login.

(continua)

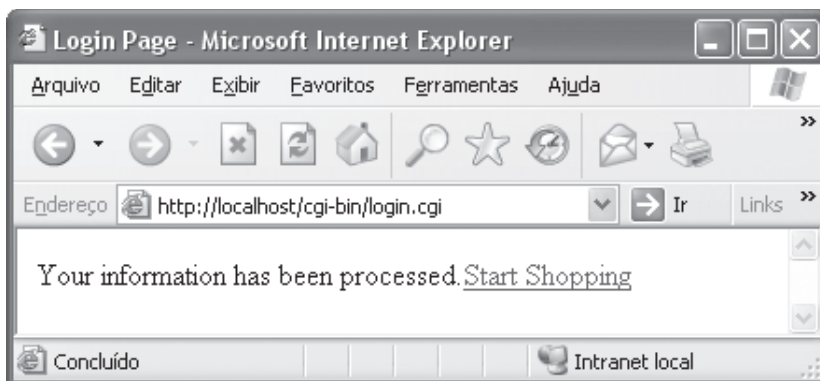
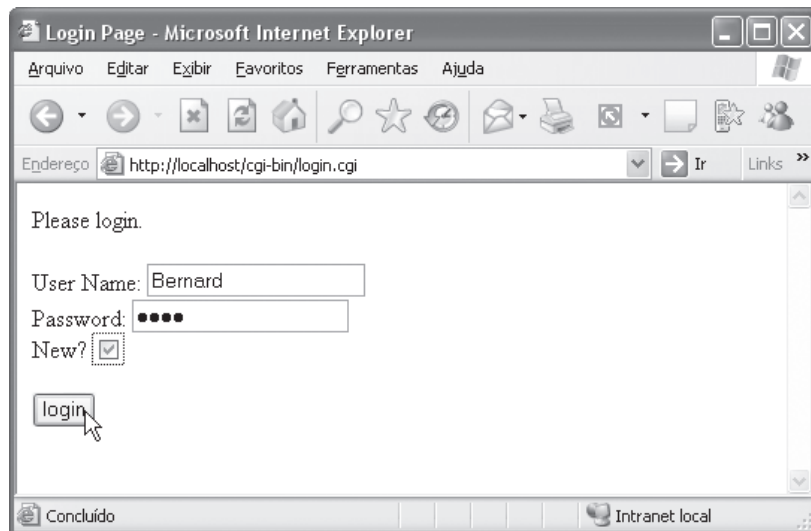
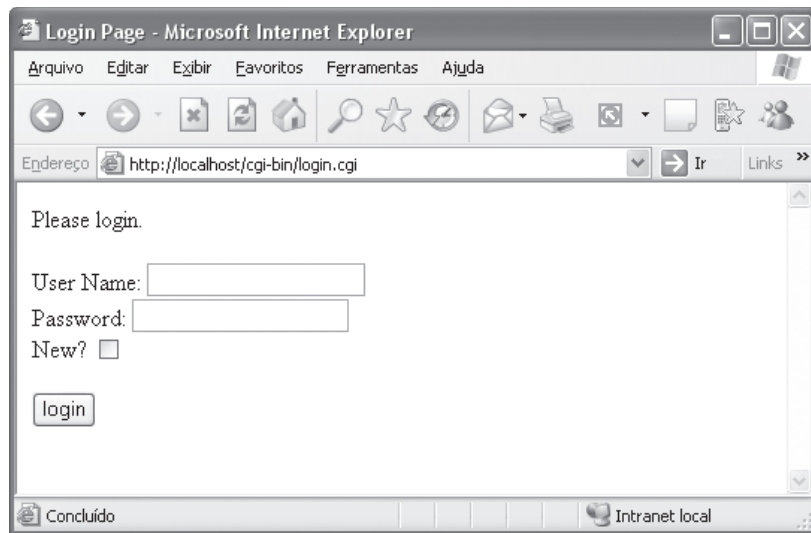
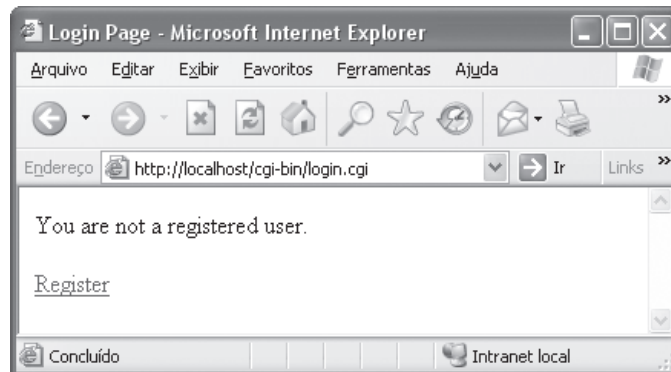
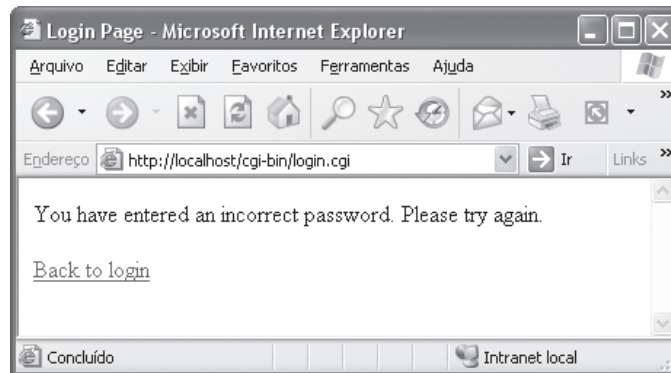
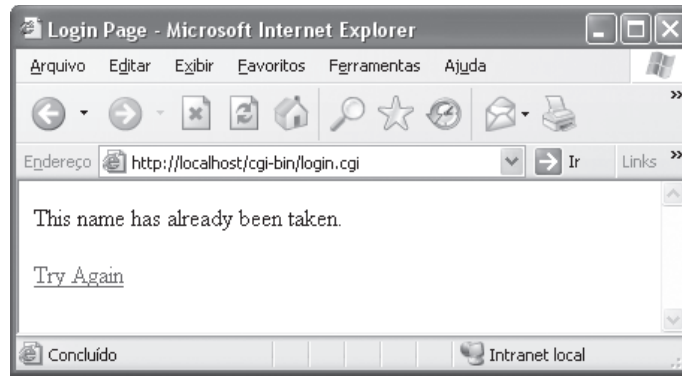


Figura 19.21 Programa que gera saída de uma página de login.

(continua)



**Figura 19.21** Programa que gera saída de uma página de login.

(continuação)

A Figura 19.22 utiliza os valores em `catalog.txt` (Figura 19.25) para a entrada em uma tabela de XHTML dos itens que o usuário pode comprar (linhas 45–82). A última coluna de cada linha inclui um botão para adicionar o item ao carrinho de compras. As linhas 63–65 geram saída dos diferentes valores de cada livro, e as linhas 71–76 geram saída de um formulário que contém o botão submit para adicionar cada livro ao carrinho de compras. Os campos de formulário ocultos são especificados para cada livro e suas informações associadas. Observe que o documento XHTML resultante enviado para o cliente contém vários formulários, um para cada livro. Entretanto, o usuário pode enviar somente um formulário por vez. Os pares nome-valor dos campos ocultos dentro do formulário enviado são postados para o script `viewcart.cgi`.

Quando um usuário compra um livro, o script `viewcart.cgi` é solicitado e o ISBN do livro a ser comprado é enviado para o script por meio de um campo de formulário oculto. A Figura 19.23 começa lendo o valor do cookie armazenado no sistema do usuário (linha 35). Quaisquer dados de cookie existentes são armazenados na string `cookieString` (linha 36). O número de ISBN inserido a partir do formulário da Figura 19.22 é armazenado em string `isbnEntered` (linha 52). O script então determina se o carrinho já contém os dados (linha 61). Se não contiver, `cookieString` recebe o valor do número do ISBN inserido (linha 62). Se o cookie já contiver os dados, o ISBN inserido é acrescentado aos dados de cookie existentes (linha 64). O novo livro é armazenado no cookie CART nas linhas 67–68. A linha 84 gera saída do conteúdo do carrinho em uma tabela chamando a função `displayShoppingCart`.

A função `displayShoppingCart` exibe os itens no carrinho de compras em uma tabela. A linha 109 abre o arquivo do lado do servidor `catalog.txt`. Se o arquivo abrir com sucesso, as linhas 122–155 obtêm, cada uma, as informações do livro (incluindo seu título, direitos autorais, ISBN e preço) do arquivo. As linhas 125–138 armazenam essas partes de dados em objetos `string`. As linhas 140–148 contam quantas vezes o ISBN atual aparece no cookie (isto é, no carrinho de compras). Se o livro atual aparece no carrinho do usuário, as linhas 151–154 exibem uma linha de tabela contendo o título, direitos autorais, ISBN e preço do livro, bem como o número de cópias do livro que o usuário escolheu comprar.

```

1 // Figura 19.22: shop.cpp
2 // Programa para exibir livros disponíveis.
3 #include <iostream>
4 using std::cerr;
5 using std::cout;
6 using std::ios;
7
8 #include <fstream>
9 using std::ifstream;
10
11 #include <string>
12 using std::string;
13
14 #include <cstdlib>
15 using std::exit;
16
17 void header();
18
19 int main()
20 {
21 // variáveis para armazenar informações de produto
22 char book[50] = "";
23 char year[50] = "";
24 char isbn[50] = "";
25 char price[50] = "";
26
27 string bookString = "";
28 string yearString = "";
29 string isbnString = "";
30 string priceString = "";
31
32 ifstream userData("catalog.txt", ios::in); // abre arquivo para entrada
33
34 // o arquivo não pôde ser aberto
35 if (!userData)
36 {
37 cerr << "Could not open database.";
38 exit(1);
39 } // fim do if
40
41 header(); // gera saída do cabeçalho
42
43 // gera saída de livros disponíveis
44 cout << "<center>
Books available for sale

"
45 << "<table border = \"1\" cellpadding = \"7\" >";
46
47 // o arquivo está aberto
48 while (userData)

```

**Figura 19.22** Script CGI que permite aos usuários comprar um livro.

(continua)

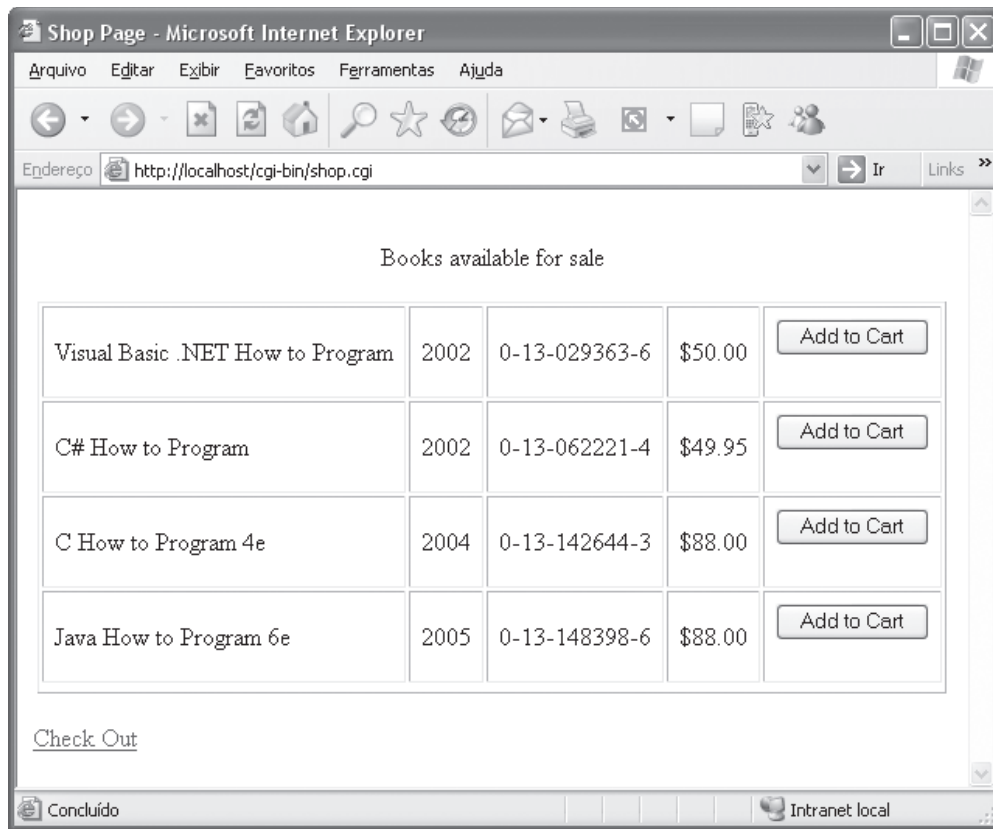
```

49 {
50 // recupera dados de arquivo
51 userData.getline(book, 50);
52 bookString = book;
53
54 userData.getline(year, 50);
55 yearString = year;
56
57 userData.getline(isbn, 50);
58 isbnString = isbn;
59
60 userData.getline(price, 50);
61 priceString = price;
62
63 cout << "<tr><td>" << bookString << "</td><td>" << yearString
64 << "</td><td>" << isbnString << "</td><td>" << priceString
65 << "</td>";
66
67 // arquivo ainda está aberto depois das leituras
68 if (userData)
69 {
70 // gera saída de formulário com botão de compra
71 cout << "<td><form method=\"post\" "
72 << "action=\"/cgi-bin/viewcart.cgi\">"
73 << "<input type=\"hidden\" name=\"add\" value=\"true\"/>"
74 << "<input type=\"hidden\" name=\"isbn\" value=\""
75 << isbnString << "\"/>" << "<input type=\"submit\" "
76 << "value=\"Add to Cart\"/>\n</form></td>\n";
77 } // fim do if
78
79 cout << "</tr>\n";
80 } // fim do while
81
82 cout << "</table></center>
"
83 << "Check Out"
84 << "</body></html>";
85 return 0;
86 } // fim de main
87
88 // função para gerar saída de informações de cabeçalho
89 void header()
90 {
91 cout << "Content-Type: text/html\n\n"; // gera saída do cabeçalho
92
93 // gera saída da declaração XML e DOCTYPE
94 cout << "<?xml version = \"1.0\"?>"
95 << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.1//EN\" "
96 << "\"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd\">";
97
98 // gera saída do elemento html e parte de seu conteúdo
99 cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
100 << "<head><title>Shop Page</title></head><body>";
101 } // fim da função header

```

**Figura 19.22** Script CGI que permite aos usuários comprar um livro.

(continua)



**Figura 19.22** Script CGI que permite aos usuários comprar um livro.

(continuação)

```

1 // Figura 19.23: viewcart.cpp
2 // Programa para ver livros no carrinho de compras.
3 #include <iostream>
4 using std::cerr;
5 using std::cin;
6 using std::cout;
7 using std::ios;
8
9 #include <fstream>
10 using std::ifstream;
11
12 #include <string>
13 using std::string;
14
15 #include <cstdlib>
16 using std::getenv;
17 using std::atoi;
18 using std::exit;
19
20 void displayShoppingCart(const string &);
21
22 int main()
23 {
24 char query[1024] = ""; // variável para armazenar string de consulta

```

**Figura 19.23** Script CGI que permite aos usuários ver o conteúdo dos carrinhos.

(continua)

```

25 string cartData; // variável a armazenar conteúdo do carrinho
26
27 string dataString = "";
28 string cookieString = "";
29 string isbnEntered = "";
30 int contentLength = 0;
31
32 // recupera dados do cookie
33 if (getenv("HTTP_COOKIE"))
34 {
35 cartData = getenv("HTTP_COOKIE");
36 cookieString = cartData;
37 } // fim do if
38
39 // dados foram inseridos
40 if (getenv("CONTENT_LENGTH"))
41 {
42 contentLength = atoi(getenv("CONTENT_LENGTH"));
43 cin.read(query, contentLength);
44 dataString = query;
45
46 // encontra a localização do valor do isbn
47 int addLocation = dataString.find("add=") + 4;
48 int endAdd = dataString.find("&isbn");
49 int isbnLocation = dataString.find("isbn=") + 5;
50
51 // recupera número do isbn para adicionar ao carrinho
52 isbnEntered = dataString.substr(isbnLocation);
53
54 // grava o cookie
55 string expires = "Friday, 14-MAY-10 16:00:00 GMT";
56 int cartLocation = cookieString.find("CART=") + 5;
57
58 if (cartLocation > 4) // o cookie existe
59 cookieString = cookieString.substr(cartLocation);
60
61 if (cookieString == "") // não existem dados de cookie
62 cookieString = isbnEntered;
63 else // há dados de cookie
64 cookieString += "," + isbnEntered;
65
66 // configura o cookie
67 cout << "Set-Cookie: CART=" << cookieString << "; expires="
68 << expires << "; path=\n";
69 } // fim do if
70
71 cout << "Content-Type: text/html\n\n"; // gera saída do cabeçalho HTTP
72
73 // gera saída da declaração XML e DOCTYPE
74 cout << "<?xml version = \"1.0\"?>"
75 << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.1//EN\" "
76 << "\"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd\">";
77
78 // gera saída do elemento html e parte de seu conteúdo
79 cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
80 << "<head><title>Shopping Cart</title></head>"

```

**Figura 19.23** Script CGI que permite aos usuários ver o conteúdo dos carrinhos.

(continua)



```

81 << "<body><center><p>Here is your current order:</p>";
82
83 if (cookieString != "") // há dados de cookie
84 displayShoppingCart(cookieString);
85 else
86 cout << "The shopping cart is empty.";
87
88 // gera saída de links de volta para a lista de livros e para fazer o pagamento
89 cout << "</center>
";
90 cout << "Back to book list
";
91 cout << "Check Out";
92 cout << "</body></html>\n";
93 return 0;
94 } // fim de main
95
96 // função para exibir itens no carrinho de compras
97 void displayShoppingCart(const string &cookieRef)
98 {
99 char book[50] = "";
100 char year[50] = "";
101 char isbn[50] = "";
102 char price[50] = "";
103
104 string bookString = "";
105 string yearString = "";
106 string isbnString = "";
107 string priceString = "";
108
109 ifstream userData("catalog.txt", ios::in); // abre arquivo para entrada
110
111 if (!userData) // o arquivo não pôde ser aberto
112 {
113 cerr << "Could not open database.";
114 exit(1);
115 } // fim do if
116
117 cout << "<table border = 1 cellpadding = 7 >";
118 cout << "<tr><td>Title</td><td>Copyright</td><td>ISBN</td>"
119 << "<td>Price</td><td>Count</td></tr>";
120
121 // o arquivo está aberto
122 while (!userData.eof())
123 {
124 // recupera informações do livro
125 userData.getline(book, 50);
126 bookString = book;
127
128 // recupera informações do ano
129 userData.getline(year, 50);
130 yearString = year;
131
132 // recupera o número do isbn
133 userData.getline(isbn, 50);
134 isbnString = isbn;
135
136 // recupera o preço

```

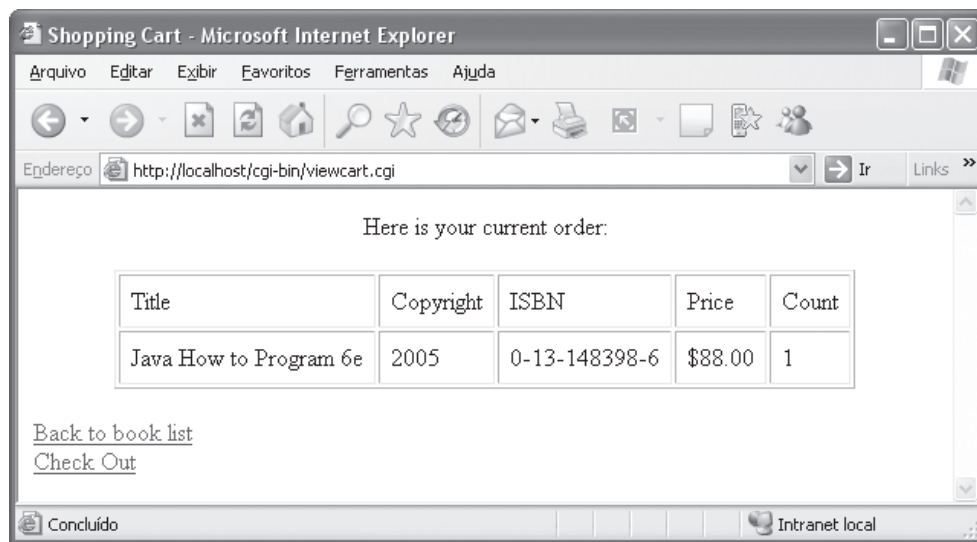
**Figura 19.23** Script CGI que permite aos usuários ver o conteúdo dos carrinhos.

(continua)

```

137 userData.getline(price, 50);
138 priceString = price;
139
140 int match = cookieRef.find(isbnString, 0);
141 int count = 0;
142
143 // correspondência foi feita
144 while (match >= 0 && isbnString != "")
145 {
146 count++;
147 match = cookieRef.find(isbnString, match + 13);
148 } // fim do while
149
150 // gera saída da linha de tabela informações do livro
151 if (count != 0)
152 cout << "<tr><td>" << bookString << "</td><td>" << yearString
153 << "</td><td>" << isbnString << "</td><td>" << priceString
154 << "</td><td>" << count << "</td></tr>";
155 } // fim do while
156
157 cout << "</table>"; // fim da tabela
158 } // fim da função displayShoppingCart

```



**Figura 19.23** Script CGI que permite aos usuários ver o conteúdo dos carrinhos.

(continuação)

A Figura 19.24 é a página que é exibida quando o usuário escolhe fazer *check out* (isto é, concluir a compra dos livros adicionados ao carrinho de compras). Esse script gera saída de uma mensagem para o usuário e chama `writeCookie` (linha 13), que apaga efetivamente as informações atuais do carrinho de compras.

A Figura 19.25 mostra o conteúdo do arquivo `catalog.txt`. Esse arquivo deve residir no mesmo diretório dos scripts CGI para que esse aplicativo de carrinho de compras funcione corretamente.

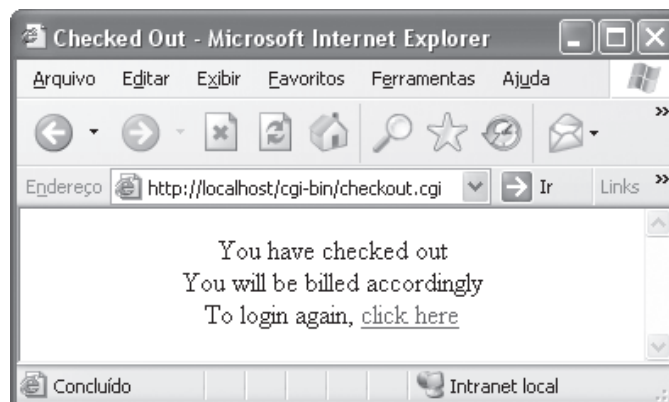
## 19.17 Síntese

Neste capítulo, discutimos o Hypertext Transfer Protocol (HTTP) e vários tipos de solicitação HTTP que especificam como um cliente faz solicitações a partir de um servidor. Você aprendeu que um aplicativo de três camadas contém uma camada de informações (também chamada de camada de dados ou camada inferior), uma camada intermediária (também chamada de lógica do negócio) e uma camada de cliente (também chamada de camada superior). Também discutimos como um navegador Web é capaz de localizar documentos em um servidor Web em resposta a solicitações de cliente. Você então aprendeu como o Common Gateway Interface (CGI) permite aos

```

1 // Figura 19.24: checkout.cpp
2 // Programa para efetuar logout do sistema.
3 #include <iostream>
4 using std::cout;
5
6 #include <string>
7 using std::string;
8
9 void writeCookie();
10
11 int main()
12 {
13 writeCookie(); // grava o cookie
14 cout << "Content-Type: text/html\n\n"; // gera saída do cabeçalho
15
16 // gera saída da declaração XML e DOCTYPE
17 cout << "<?xml version = \"1.0\"?>"
18 << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.1//EN\" "
19 << "\"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd\">";
20
21 // gera saída do elemento html e de seu conteúdo
22 cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
23 << "<head><title>Checked Out</title></head><body><center>"
24 << "<p>You have checked out
"
25 << "You will be billed accordingly
To login again, "
26 << "click here"
27 << "</center></body></html>\n";
28 return 0;
29 } // fim de main
30
31 // função para gravar o cookie
32 void writeCookie()
33 {
34 // string contendo data de expiração
35 string expires = "Friday, 14-MAY-10 16:00:00 GMT";
36
37 // configura o cookie
38 cout << "Set-Cookie: CART=; expires=" << expires << "; path=\n";
39 } // fim de writeCookie

```



**Figura 19.24** Programa de check out.

```

Visual Basic .NET How to Program
2002
0-13-029363-6
$50.00
C# How to Program
2002
0-13-062221-4
$49.95
C How to Program 4e
2004
0-13-142644-3
$88.00
Java How to Program 6e
2005
0-13-148398-6
$88.00

```

**Figura 19.25** Conteúdo de catalog.txt.

aplicativos interagir com servidores Web e clientes (por exemplo, navegadores Web). Demonstramos como gravar scripts CGI em C++ e apresentamos scripts CGI que exibem a hora local e as variáveis de ambiente de sistema, lêem dados passados pela string de consulta e lêem os dados passados por um formulário XHTML. Apresentamos também um estudo de caso que fornece um portal interativo para o site Web fictício da Bug2Bug Travel. Esse estudo de caso contém tanto documentos XHTML estáticos como scripts CGI que geram conteúdo Web dinâmico. Você aprendeu a utilizar scripts CGI para ler e gravar cookies para manter informações de estado. Então discutimos o uso de arquivos do lado do servidor como outro modo de manter informações de estado. Por fim, apresentamos um estudo de caso que controla acesso de usuário utilizando arquivo do lado do servidor e monitora compras de usuários com um carrinho de compras baseado em cookie.

No próximo capítulo discutiremos o algoritmo de pesquisa binária e o algoritmo de classificação por intercalação. Também utilizaremos a notação O maiúsculo para analisar e comparar a eficiência de vários algoritmos de pesquisa e classificação.

## 19.18 Recursos na Internet e na Web

### *Apache*

[httpd.apache.org](http://httpd.apache.org)

Essa é a home page de produtos do Apache HTTP Server. Os usuários podem fazer download do Apache a partir desse site.

[www.apacheweek.com](http://www.apacheweek.com)

Essa revista on-line contém artigos sobre trabalhos do Apache, revisões de produto e outras informações relativas ao software Apache.

[linuxtoday.com/stories/18780.html](http://linuxtoday.com/stories/18780.html)

Esse site contém um artigo sobre o Apache HTTP Server e as plataformas que o suportam. Também contém links para outros artigos sobre o Apache.

### *CGI*

[www.gnu.org/software/cgicc/cgicc.html](http://www.gnu.org/software/cgicc/cgicc.html)

Esse site contém uma biblioteca CGI de código-fonte aberto livre para criação de scripts CGI em C++.

[www.hotscripts.com](http://www.hotscripts.com)

Esse site contém uma rica coleção de scripts para realizar manipulação de imagem, administração de servidor, rede etc. utilizando CGI.

[www.jmarshall.com/easy/cgi](http://www.jmarshall.com/easy/cgi)

Essa página contém uma breve explicação sobre CGI para aqueles com experiência em programação.

[www.w3.org/CGI](http://www.w3.org/CGI)

Essa página da World Wide Web Consortium discute questões de segurança relacionadas com CGI.

[www.w3.org/Protocols](http://www.w3.org/Protocols)

Esse site do World Wide Web Consortium contém informações sobre a especificação HTTP e links para notícias, listas de mala direta e artigos publicados.

## Resumo

- Um servidor Web responde às solicitações de cliente (por exemplo, navegador Web) fornecendo recursos (por exemplo, documentos XHTML).
- O Hypertext Transfer Protocol (HTTP) é um protocolo independente de plataforma para transferir solicitações e arquivos pela Internet (isto é, entre servidores Web e navegadores Web).
- O HTTP define vários tipos de solicitação (também conhecidas como métodos de solicitação), cada um dos quais especifica como um cliente envia solicitações para um servidor.
- Os dois tipos mais comuns de solicitação HTTP são `get` e `post`. Esses tipos de solicitação recuperam e enviam dados de formulário do cliente a partir de e para um servidor Web.
- Um `form` é um elemento XHTML que pode conter campos de texto, botões de opção, caixas de seleção e outros componentes de interface gráfica com o usuário que permitem aos usuários inserir e enviar dados em uma página Web.
- Uma solicitação `get` obtém (*gets*, ou recupera) informações de um servidor. Essas solicitações freqüentemente recuperam um documento HTML ou uma imagem.
- Uma solicitação `post` posta (ou envia) dados a um servidor, como informações de autenticação ou dados de um formulário que coleta a entrada de usuário.
- Normalmente, solicitações `post` são utilizadas para postar uma mensagem em um grupo de notícias ou em um fórum de discussão, passam a entrada do usuário para um processo de tratamento de dados e armazenam ou atualizam os dados em um servidor.
- Uma solicitação HTTP freqüentemente envia dados para o handler de formulário do lado do servidor — um programa que reside no servidor Web e é criado por um programador do lado do servidor para tratar solicitações de cliente.
- Os navegadores freqüentemente armazenam em cache (isto é, salvam em um disco local) as páginas Web para recarregar rapidamente, reduzindo a quantidade de dados de download que o navegador precisa fazer pela Internet.
- Um servidor Web faz parte de um aplicativo de múltiplas camadas, às vezes referido como um aplicativo de *n*-camadas.
- Aplicativos de múltiplas camadas dividem as funcionalidades em camadas separadas (isto é, agrupamentos lógicos de funcionalidade). As camadas podem ser localizadas no mesmo computador ou em computadores separados.
- A camada de informações (também chamada de camada de dados ou camada inferior) mantém dados para o aplicativo. Em geral, a camada de informações armazena dados em um sistema de gerenciamento de banco de dados relacional (*relational database management system* – RDBMS).
- A camada intermediária (também chamada de camada da lógica do negócio) implementa a lógica do negócio para controlar interações entre clientes de aplicativo e dados de aplicativo.
- A lógica do negócio na camada intermediária impõe regras de negócio e assegura que os dados são confiáveis antes de atualizar o banco de dados ou apresentá-los para um usuário.
- A camada de cliente, ou camada superior, é a interface com o usuário do aplicativo.
- Os usuários podem solicitar documentos a partir de servidores Web locais (isto é, aqueles que residem em máquinas de usuários) ou servidores Web remotos (isto é, aqueles que residem em máquinas em uma rede).
- Os servidores Web locais podem ser acessados de duas maneiras: pelo nome de máquina ou pelo `localhost` — um nome de host que referencia a máquina local.
- Um host é um computador que armazena e mantém recursos, como páginas Web, bancos de dados e arquivos multimídia.
- Um domínio representa um grupo de hosts na Internet.
- Cada domínio tem um nome de domínio, também conhecido como endereço Web, que identifica de maneira exclusiva a localização de um negócio ou organização na Internet.
- Um nome de domínio completamente qualificado (*fully qualified domain name* – FQDN), também conhecido como o nome de máquina, contém um host (por exemplo, `www` para World Wide Web) e um nome de domínio, incluindo um domínio de nível mais alto (*top-level domain* – TLD).
- O domínio de nível mais alto é o último e mais significativo componente de um nome de domínio completamente qualificado.
- Cada FQDN corresponde a um endereço numérico chamado de endereço IP (Internet Protocol), que é muito semelhante a um endereço residencial.
- Um servidor DNS (Domain Name System) é um computador que mantém um banco de dados de FQDNs e seus endereços IP correspondentes. O processo de converter FQDN em endereços IP é chamado de pesquisa DNS.
- Na estrutura de diretório de Apache HTTP Server, os documentos XHTML devem ser salvos no diretório `htdocs`.
- O Common Gateway Interface (CGI) é um protocolo-padrão para permitir aos aplicativos (comumente chamados programas CGI ou scripts CGI) interagir com servidores Web e (indiretamente) com clientes (por exemplo, navegadores Web).

- O CGI é freqüentemente utilizado para gerar conteúdo Web dinâmico utilizando entrada de cliente, bancos de dados e outros serviços de informações.
- Uma página Web é dinâmica se seu conteúdo for gerado programaticamente quando a página é solicitada, ao contrário do conteúdo Web estático, que não é gerado programaticamente quando a página é solicitada (isto é, a página já existe antes de a solicitação ser feita).
- O HTTP descreve um conjunto de métodos e cabeçalhos que permitem aos clientes e servidores interagir e trocar informações de maneira uniforme e previsível.
- Um documento XHTML é um arquivo de texto simples que contém marcações (elementos) que descrevem a estrutura dos dados que o documento contém.
- Os documentos XHTML também podem conter as informações sobre hipertexto (normalmente chamados hyperlinks), que são vinculados com outras páginas Web ou outras localizações na mesma página.
- Um URL contém o protocolo do recurso (como http), o nome de máquina ou endereço IP do recurso e o nome (incluindo o caminho) do recurso.
- O método HTTP get indica que o cliente deseja recuperar um recurso.
- As informações no cabeçalho Content-Type identificam o tipo MIME (*Multipurpose Internet Mail Extensions*) do conteúdo.
- Cada tipo de dados enviados a partir do servidor tem um tipo MIME pelo qual o navegador determina como processar os dados que ele recebe.
- É possível redirecionar (ou fazer *pipe*) a saída-padrão para outro destino.
- A função `time` obtém a hora atual, que é representada como o número de segundos que passou desde a meia-noite de 1ª de janeiro de 1970 e armazena o valor recuperado na localização especificada pelo parâmetro.
- A função `localtime` da biblioteca C++, quando recebe uma variável `time_t`, retorna um ponteiro para um objeto contendo a hora local dividida em campos (isto é, dias, horas etc. são colocados em membros de objeto individuais).
- A função `asctime`, que aceita um ponteiro para um objeto contendo a hora dividida em campos, retorna uma string como `Wed Oct 31 13:10:37 2004`.
- As variáveis de ambiente contêm informações sobre o ambiente do cliente e do servidor, tais como o tipo de navegador Web em uso e a localização do documento no servidor.
- O valor associado com uma variável de ambiente pode ser obtido chamando a função `getenv` de `<cstdlib>` e passando para ela o nome da variável de ambiente.
- A variável de ambiente `QUERY_STRING` contém informações que são acrescentadas a um URL em uma solicitação get.
- O elemento `form` inclui um formulário XHTML e, geralmente, aceita dois atributos. O primeiro atributo é `action`, que especifica o recurso de servidor a executar quando o usuário envia o formulário. O segundo é `method`, que identifica o tipo de solicitação HTTP (isto é, `get` ou `post`) a utilizar quando o navegador enviar o formulário para o servidor Web.
- O método `post` configura a variável de ambiente `CONTENT_LENGTH`, para indicar o número de caracteres de dados enviado em uma solicitação `post`.
- A função `atoi` de `<cstdlib>` pode ser utilizada para converter o valor contido em `CONTENT_LENGTH` em um inteiro.
- O cabeçalho `Refresh` redireciona o cliente para uma nova localização depois de uma quantidade especificada de tempo.
- O cabeçalho `Location` redireciona o cliente para uma nova localização.
- O cabeçalho `Status` instrui o servidor a gerar saída de uma linha de cabeçalho de status especificada (como `HTTP/1.1 200 OK`).
- Os cookies são arquivos de texto essencialmente pequenos que um servidor Web envia para o navegador, que então os salva no computador. Os cookies são enviados pelo navegador de volta para o servidor com cada solicitação subsequente para o mesmo caminho até que os cookies expirem ou sejam excluídos.
- O cabeçalho `Set-Cookie`: indica que o navegador deve armazenar os dados entrantes em um cookie.
- A variável de ambiente `HTTP_COOKIE` armazena os cookies do cliente.
- As informações de estado podem ser mantidas utilizando cookies ou criando arquivos do lado do servidor (isto é, arquivos que são localizados no servidor ou na rede do servidor).
- Somente alguém com acesso e permissão para alterar arquivos no servidor pode fazer isso.

## Terminologia

`action`, atributo do elemento `form`  
 Apache HTTP Server  
 aplicativo de múltiplas camadas  
 aplicativo de *n*-camadas  
 arquivo do lado do servidor

`asctime`, função  
 cabeçalho de HTTP  
 cache  
 camada  
 camada da lógica do negócio

camada de cliente  
 camada de dados  
 camada de informações  
 camada inferior  
 camada intermediária

camada superior	get, tipo de solicitação HTTP	QUERY_STRING, variável de ambiente
CGI (Common Gateway Interface)	getenv, função de <cstdlib>	Refresh, cabeçalho
CONTENT_LENGTH, variável de ambiente	handler do formulário do lado do servidor	regra de negócio
Content-Type, cabeçalho	hipertexto	script CGI
conteúdo Web dinâmico	host	servidor Web
conteúdo Web dinâmico <i>versus</i> conteúdo Web estático	htdocs, diretório	servidor Web HTTPd
conteúdo Web estático	HTTP_COOKIE, variável de ambiente	servidor Web local
cookie	hyperlink	servidor Web remoto
diretório virtual	Hypertext Transfer Protocol (HTTP)	Set-Cookie:, cabeçalho de HTTP
DNS (Domain Name System), servidor	localhost	sistema de gerenciamento de banco de dados relacional (RDBMS)
domínio	localtime, função	software de código-fonte aberto
domínio de nível mais alto ( <i>top-level domain</i> – TLD)	Location, cabeçalho	solicitação, método
elemento	lógica do negócio	Status, cabeçalho
endereço de loopback	marcações	string de consulta
endereço IP (Internet Protocol)	method, atributo do elemento form	tag final
endereço Web	Multipurpose Internet Mail Extensions (MIME)	tag inicial
Extensible Hypertext Markup Language (XHTML)	nome de domínio	time, função
form, elemento XHTML (<form>... </form>)	nome de domínio completamente qualificado (FQDN)	time_t
GET, método HTTP	pesquisa DNS	título, elemento XHTML (<title>...</title>)
	pipe	Universal Resource Locator (URL)
	post, tipo de solicitação HTTP	variável de ambiente
	programa CGI	

## Exercícios de revisão

**19.1** Preencha as lacunas em cada uma das seguintes sentenças:

- Os dois tipos mais comuns de solicitação HTTP são \_\_\_\_\_ e \_\_\_\_\_.
- Os navegadores freqüentemente \_\_\_\_\_ as páginas Web para recarregamento rápido.
- Em um aplicativo de três camadas, um servidor Web normalmente faz parte da camada \_\_\_\_\_.
- No URL `http://www.deitel.com/books/downloads.html`, `www.deitel.com` é o(a) \_\_\_\_\_ do servidor, onde um cliente pode localizar o recurso desejado.
- Um documento \_\_\_\_\_ é um arquivo de texto contendo marcações que descrevem para um navegador Web como exibir e formatar as informações no documento.
- A variável de ambiente \_\_\_\_\_ fornece um mecanismo para oferecer dados para scripts CGI.
- Uma maneira comum de ler a entrada a partir do usuário é implementar o elemento XHTML \_\_\_\_\_.

**19.2** Determine se cada uma das seguintes sentenças é *verdadeira* ou *falsa*. Se *falsa*, explique por quê.

- Os servidores Web e clientes se comunicam entre si pelo HTTP independente de plataforma.
- Os servidores Web freqüentemente armazenam em cache as páginas Web para recarregar.
- A camada de informações implementa a lógica do negócio para controlar o tipo de informação que é apresentada a um cliente particular.
- Uma página Web dinâmica é uma página Web que não é criada programaticamente.
- Colocamos dados em uma string de consulta utilizando um formato que consiste em uma série de pares nome-valor unidos por pontos de exclamação (!).
- Utilizar um script CGI é mais eficiente que utilizar um documento XHTML.
- O método `post` de enviar dados de formulário é preferível a `get` ao enviar informações pessoais para o servidor Web.

## Respostas dos exercícios de revisão

**19.1** a) `get` e `post`. b) armazenam em cache. c) intermediária. d) nome de máquina. e) XHTML. f) QUERY\_STRING. g) `form`.

**19.2** a) Verdadeira. b) Verdadeira. c) Falsa. A camada intermediária implementa a lógica do negócio para controlar interações entre clientes de aplicativo e dados de aplicativo. d) Falsa. Uma página Web dinâmica é uma página Web que é criada programaticamente. e) Falsa. Os pares são unidos por um *e* comercial (&). f) Falsa. Os documentos XHTML são mais eficientes que os scripts CGI, porque não precisam ser executados no lado do servidor antes de serem enviados para saída no cliente. g) Verdadeira.



## Exercícios

- 19.3** Defina os termos a seguir:
- a) HTTP.
  - b) Aplicativo de múltiplas camadas.
  - c) Método de solicitação.
- 19.4** Explique a diferença entre o tipo de solicitação `get` e o tipo de solicitação `post`. Quando é ideal utilizar o tipo de solicitação `post`?
- 19.5** Escreva um script CGI que imprime os quadrados dos inteiros de 1 a 10 em linhas separadas.
- 19.6** Escreva um script CGI que recebe como entrada três números a partir do cliente e exibe uma afirmação que indica se os três números poderiam representar um triângulo equilátero (os três lados têm o mesmo comprimento), um triângulo isósceles (os dois lados têm o mesmo comprimento) ou um triângulo reto (o quadrado de um lado é igual à soma dos quadrados dos outros dois lados).
- 19.7** Escreva um script de adivinhação que permite ao usuário enviar uma pergunta. Quando a pergunta é enviada, o script deve escolher uma resposta aleatória em uma lista de respostas (tal como "Pode ser", "Provavelmente não", "Certamente", "Não parece bom" e "Sim") e exibir a resposta ao cliente.
- 19.8** Modifique o programa das figuras 19.13–19.14 para incorporar o formulário XHTML de abertura e o processamento dos dados em um único script CGI (isto é, combinar a XHTML da Figura 19.13 com o script CGI da Figura 19.14). Quando o script CGI é inicialmente solicitado, o formulário deve ser exibido. Quando o formulário é enviado, o script CGI deve exibir o resultado.
- 19.9** Modifique o script `viewcart.cgi` (Figura 19.23) para permitir aos usuários remover alguns itens do carrinho de compras.



*Entre lágrimas e soluções,  
escolheu os de maior  
tamanho...*

Lewis Carroll

*Tente o último e nunca duvides;  
Nada é tão difícil, mas a  
pesquisa o encontrará.*

Robert Herrick

*Guardei-o na memória, e a  
chave a levas.*

William Shakespeare

*É uma lei imutável nos negócios  
que palavras são palavras,  
explicações são explicações,  
promessas são promessas  
— mas somente desempenho é  
realidade.*

Harold S. Green

## Pesquisa e classificação

### OBJETIVOS

Neste capítulo, você aprenderá:

- A procurar um dado valor em um vetor utilizando a pesquisa binária.
- A classificar um vetor utilizando o algoritmo de classificação por intercalação recursiva.
- A determinar a eficiência dos algoritmos de pesquisa e de classificação.

- 20.1** Introdução
- 20.2** Algoritmos de pesquisa
  - 20.2.1** Eficiência da pesquisa linear
  - 20.2.2** Pesquisa binária
- 20.3** Algoritmos de classificação
  - 20.3.1** Eficiência da classificação por seleção
  - 20.3.2** Eficiência da classificação por inserção
  - 20.3.3** Classificação por intercalação (uma implementação recursiva)
- 20.4** Síntese

Resumo | Terminologia | Exercícios de revisão | Respostas dos exercícios de revisão | Exercícios

## 20.1 Introdução

**Pesquisar** dados envolve determinar se um valor (chamado **chave de pesquisa**) está presente nos dados e, se estiver, encontrar a localização do valor. Dois algoritmos de pesquisa populares são a pesquisa linear simples (introduzida na Seção 7.7) e a pesquisa binária mais rápida, mas mais complexa, que é introduzida neste capítulo.

**Classificar** coloca os dados na ordem, em geral ascendente ou descendente, com base em uma ou mais **chaves de classificação**. Uma lista de nomes poderia ser classificada alfabeticamente, contas bancárias poderiam ser classificadas pelo número de conta, registros de folha de pagamento de funcionários poderiam ser classificados pelo CIC e assim por diante. Anteriormente, você aprendeu a classificação por inserção (Seção 7.8) e por seleção (Seção 8.6). Este capítulo introduz a classificação por intercalação, que é mais eficiente, porém mais complexa. A Figura 20.1 resume os algoritmos de pesquisa e de classificação discutidos nos exemplos e exercícios deste livro. Este

Capítulo	Algoritmo	Posição
<i>Algoritmos de pesquisa</i>		
7	Pesquisa linear	Seção 7.7
20	Pesquisa binária	Seção 20.2.2
	Pesquisa linear recursiva	Exercício 20.8
	Pesquisa binária recursiva	Exercício 20.9
	Pesquisa em árvore binária	Seção 21.7
21	Pesquisa linear de uma lista vinculada	Exercício 21.21
	Função <code>binary_search</code> da biblioteca-padrão	Seção 23.5.6
<i>Algoritmos de classificação</i>		
7	Classificação por inserção	Seção 7.8
8	Classificação por seleção	Seção 8.6
20	Classificação por intercalação recursiva	Seção 20.3.3
	Classificação por borbulhamento ( <i>bubble sort</i> )	Exercícios 20.5 e 20.6
	<i>Bucket sort</i>	Exercício 20.7
	<i>Quicksort</i> recursivo	Exercício 20.10
21	Classificação em árvore binária	Seção 21.7
23	Função <code>sort</code> da biblioteca-padrão	Seção 23.5.6
	Classificação por heap	Seção 23.5.12

**Figura 20.1** Algoritmos de pesquisa e classificação neste texto.

capítulo também introduz a **notação O**, que é utilizada para estimar o tempo de execução de pior cenário para um algoritmo — isto é, o grau de dificuldade de funcionamento que um algoritmo pode apresentar para resolver um problema.

## 20.2 Algoritmos de pesquisa

Pesquisar um número de telefone, acessar um site Web e verificar a definição de uma palavra em um dicionário, todas essas operações envolvem pesquisar grandes volumes de dados. Todos os algoritmos de pesquisa têm o mesmo objetivo — localizar um elemento que corresponde a uma dada chave de pesquisa, se esse elemento, de fato, existir. Mas há alguns aspectos que diferenciam os algoritmos de pesquisa entre si. A principal diferença é o esforço que eles exigem para completar a pesquisa. Uma maneira de descrever esse esforço é com a notação O. Para algoritmos de pesquisa e classificação, isso depende particularmente do número de elementos de dados.

No Capítulo 7, discutimos o algoritmo de pesquisa linear, que é um algoritmo de pesquisa simples e fácil de implementar. Agora discutiremos a eficiência do algoritmo de pesquisa linear medida pela notação O. Então, introduziremos um algoritmo de pesquisa que é relativamente eficiente, mas mais complexo e difícil de implementar.

### 20.2.1 Eficiência da pesquisa linear

Suponha que um algoritmo simplesmente testa se o primeiro elemento de um vetor é igual ou não ao segundo elemento do vetor. Se o vetor tiver 10 elementos, esse algoritmo exigirá uma comparação. Se o vetor tiver 1.000 elementos, mesmo assim o algoritmo exigirá uma comparação. Na realidade, o algoritmo é completamente independente do número de elementos no vetor. Diz-se que esse algoritmo tem um **tempo de execução constante**, representado na notação O como **O(1)**. Um algoritmo que é **O(1)** não necessariamente exige somente uma comparação. **O(1)** simplesmente significa que o número de comparações é *constante* — não aumenta à medida que o tamanho do vetor aumenta. Um algoritmo que testa se o primeiro elemento de um vetor é igual a qualquer um dos três elementos seguintes sempre exigirá três comparações, mas na notação O esse algoritmo é considerado **O(1)**. **O(1)** é freqüentemente pronunciado ‘na ordem de 1’ ou mais simplesmente ‘**ordem 1**’.

Um algoritmo que testa se o primeiro elemento de um vetor é igual a *qualquer* um dos outros elementos do vetor requer no máximo  $n - 1$  comparações, onde  $n$  é o número de elementos no vetor. Se o vetor tiver 10 elementos, esse algoritmo exigirá até nove comparações. Se o vetor tiver 1.000 elementos, esse algoritmo exigirá até 999 comparações. À medida que  $n$  aumenta, a parte  $n$  da expressão ‘predomina’ e subtrair um torna-se irrelevante. A notação O é projetada para destacar esses termos dominantes e ignorar termos que não têm importância à medida que  $n$  aumenta. Por essa razão, diz-se que um algoritmo que exige um total de  $n - 1$  comparações (como aquele descrito neste parágrafo) é **O(n)**. Diz-se que um algoritmo **O(n)** tem um **tempo de execução linear**. **O(n)** costuma ser pronunciado ‘na ordem de  $n$ ’ ou mais simplesmente ‘**ordem n**’.

Agora suponha que você tem um algoritmo que testa se *qualquer* elemento de um vetor é duplicado em outra parte do vetor. O primeiro elemento deve ser comparado a cada um dos elementos no vetor. O segundo elemento deve ser comparado a todos os outros elementos, exceto o primeiro (ele já foi comparado ao primeiro). O terceiro elemento deve ser comparado a todos os outros elementos, exceto os dois primeiros. No final, esse algoritmo terminará fazendo  $(n - 1) + (n - 2) + \dots + 2 + 1$  ou  $n^2/2 - n/2$  comparações. À medida que  $n$  aumenta, o termo  $n^2$  predomina e o termo  $n$  torna-se irrelevante. Mais uma vez, a notação O destaca o termo  $n^2$ , deixando  $n^2/2$ . Mas como veremos a seguir, fatores constantes são omitidos na notação O.

A notação O se preocupa em como o tempo de execução de um algoritmo aumenta em relação ao número de itens processado. Suponha que um algoritmo exija  $n^2$  comparações. Com quatro elementos, o algoritmo exigirá 16 comparações; com oito elementos, 64 comparações. Com esse algoritmo, dobrar o número de elementos quadruplica o número de comparações. Considere um algoritmo semelhante que exige  $n^2/2$  comparações. Com quatro elementos, o algoritmo exigirá oito comparações; com oito elementos, 32 comparações. Mais uma vez, dobrar o número de elementos quadruplica o número de comparações. Esses dois algoritmos aumentam conforme o quadrado de  $n$ , assim a notação O ignora a constante e os dois algoritmos são considerados como **O(n<sup>2</sup>)**, o que é chamado **tempo de execução quadrático**, pronunciado como ‘na ordem de  $n$  ao quadrado’ ou mais simplesmente ‘**ordem do quadrado de  $n$** ’.

Quando  $n$  é pequeno, algoritmos **O(n<sup>2</sup>)** (executando um bilhão de operações por segundo nos atuais computadores pessoais) não afetarão significativamente o desempenho. Mas, à medida que  $n$  aumenta, você começa a observar a degradação no desempenho. Um algoritmo **O(n<sup>2</sup>)** executando em um vetor de um milhão de elementos exigiria um trilhão de ‘operações’ (onde cada uma na verdade exigiria várias instruções de máquina para executar). Isso consumiria algumas horas para executar. Um vetor de um bilhão de elementos exigiria um quintilhão de operações, um número tão grande que o algoritmo demoraria décadas para executar! Os algoritmos **O(n<sup>2</sup>)** são fáceis de escrever, como você viu nos capítulos anteriores. Neste capítulo, você verá algoritmos com medidas O mais favoráveis. Freqüentemente, esses eficientes algoritmos demandam um pouco mais de inteligência e esforço para serem criados, mas seu desempenho superior recompensa muito bem o esforço extra, especialmente à medida que o  $n$  torna-se grande e algoritmos são compostos em programas maiores.

O algoritmo de pesquisa linear executa em tempo **O(n)**. O pior caso nesse algoritmo é que cada elemento deve ser verificado para determinar se a chave de pesquisa existe no vetor. Se o tamanho do vetor for dobrado, o número de comparações que o algoritmo deve realizar também será dobrado. Observe que a pesquisa linear pode fornecer um excelente desempenho se o elemento que corresponde à chave de pesquisa estiver próximo ou no início do vetor. Mas buscamos algoritmos que, na média, tenham um bom desempenho em todas as pesquisas, incluindo aqueles em que o elemento correspondente à chave de pesquisa está próximo do final do vetor.

A pesquisa linear é o algoritmo de pesquisa mais fácil de implementar, mas pode ser lento se comparado com outros algoritmos de pesquisa. Se um programa precisar realizar muitas pesquisas em grandes vetores, será melhor implementar um algoritmo diferente, mais eficiente, como a pesquisa binária que apresentaremos na próxima seção.



## Dica de desempenho 20.1

Às vezes os algoritmos mais simples demonstram um desempenho pobre. Sua virtude é que eles são fáceis de programar, testar e depurar. Às vezes, são necessários algoritmos mais complexos para conseguir desempenho máximo.

### 20.2.2 Pesquisa binária

O **algoritmo de pesquisa binária** é mais eficiente que o algoritmo de pesquisa linear, mas exige que primeiro o vetor seja classificado. Isso só é vantajoso quando o vetor, uma vez classificado, for pesquisado várias vezes — ou quando o aplicativo de pesquisa tiver rigorosos requisitos de desempenho. A primeira iteração desse algoritmo testa o elemento no meio do vetor. Se isso corresponder à chave de pesquisa, o algoritmo termina. Supondo que o vetor seja classificado em ordem crescente, se a chave de pesquisa for menor que o elemento do meio, a chave de pesquisa não poderá localizar nenhum elemento na segunda metade do vetor e o algoritmo continuará com apenas a primeira metade do vetor (isto é, até o primeiro elemento, mas sem incluir o elemento do meio). Se a chave de pesquisa for maior que o elemento no meio, a chave de pesquisa não poderá localizar nenhum elemento na primeira metade do vetor e o algoritmo continuará apenas com a segunda metade do vetor (isto é, o elemento depois do elemento do meio até o último elemento). Cada iteração testa o valor do meio da parte restante do vetor. Se o elemento não corresponder à chave de pesquisa, o algoritmo elimina metade dos elementos restantes. O algoritmo termina localizando um elemento que corresponda à chave de pesquisa ou reduzindo o subvetor ao tamanho zero.

Como um exemplo, considere o vetor de 15 elementos classificados

2 3 5 10 27 30 34 51 56 65 77 81 82 93 99

e uma chave de pesquisa de 65. Um programa que implementa o algoritmo de pesquisa binária primeiro verificaria se 51 é a chave de pesquisa (uma vez que 51 é o elemento no meio do vetor). A chave de pesquisa (65) é maior que 51, assim 51 é descartado junto com a primeira metade do vetor (todos os elementos menores que 51). Em seguida, o algoritmo verifica se 81 (o elemento no meio do restante do vetor) corresponde à chave de pesquisa. A chave de pesquisa (65) é menor que 81, portanto 81 é descartado junto com os elementos maiores que 81. Depois de apenas dois testes, o algoritmo reduziu a três o número de elementos a verificar (56, 65 e 77). O algoritmo então verifica 65 (que, de fato, corresponde à chave de pesquisa) e retorna o índice (9) do elemento do vetor contendo 65. Nesse caso, o algoritmo exigiu somente três comparações para determinar se um elemento do vetor correspondia à chave de pesquisa. Utilizar um algoritmo de pesquisa linear exigiria 10 comparações. [Nota: Neste exemplo, escolhemos utilizar um vetor com 15 elementos, assim haverá sempre um elemento intermediário óbvio no vetor. Com um número par de elementos, o meio do vetor reside entre dois elementos. Implementamos o algoritmo para escolher o maior desses dois elementos.]

As figuras 20.2–20.3 definem a classe `BinarySearch` e suas funções-membro, respectivamente. A classe `BinarySearch` é semelhante a `LinearSearch` (Seção 7.7) — ela tem um construtor, uma função de pesquisa (`binarySearch`), uma função `displayElements`, dois membros de dados `private` e uma função utilitária `private` (`displaySubElements`). As linhas 18–28 da Figura 20.3 definem o construtor. Depois de inicializar o vetor com `ints` aleatórios de 10–99 (linhas 24–25), a linha 27 chama a função `Standard Library sort` no vetor `data`. A função `sort` aceita dois **iteradores de acesso aleatório** e classifica os elementos no vetor `data` em ordem crescente. Um iterador de acesso aleatório é um iterador que permite acesso a qualquer item de dados no vetor a qualquer hora. Neste caso, utilizamos `data.begin()` e `data.end()` para incluir todo o vetor. Lembre-se de que o algoritmo de pesquisa binária só funcionará em um vetor classificado.

```

1 // Figura 20.2: BinarySearch.h
2 // Classe que contém um vetor de inteiros aleatórios e uma função
3 // que utiliza a pesquisa binária para localizar um inteiro.
4 #include <vector>
5 using std::vector;
6
7 class BinarySearch
8 {
9 public:
10 BinarySearch(int); // construtor inicializa vetor
11 int binarySearch(int) const; // realiza uma pesquisa binária no vetor
12 void displayElements() const; // exibe elementos do vetor
13 private:
14 int size; // tamanho do vetor
15 vector< int > data; // vetor de ints
16 void displaySubElements(int, int) const; // exibe intervalo de valores
17 }; // fim da classe BinarySearch

```

**Figura 20.2** Definição de classe `BinarySearch`.

As linhas 31–61 definem a função `binarySearch`. A chave de pesquisa é passada para o parâmetro `searchElement` (linha 31). As linhas 33–35 calculam o índice da extremidade `low`, o índice da extremidade `high` e o índice `middle` da parte do vetor em que o programa está atualmente pesquisando. No início da função, a extremidade `low` é 0, a extremidade `high` é o tamanho do vetor menos 1 e `middle` é a média desses dois valores. A linha 36 inicializa a `location` do elemento localizado como `-1` — o valor que será retornado se a chave de pesquisa não for encontrada. As linhas 38–58 fazem um loop até que `low` seja maior que `high` (isso ocorre quando o elemento não é localizado) ou a `location` não é igual a `-1` (indicando que a chave de pesquisa foi localizada). A linha 50 testa se o valor no elemento `middle` é igual a `searchElement`. Se isso for `true`, a linha 51 atribui `middle` a `location`. Então, o loop termina e `location` é retornada ao chamador. Cada iteração do loop testa um único valor (linha 50) e elimina metade dos valores restantes no vetor (linha 53 ou 55).

```

1 // Figura 20.3: BinarySearch.cpp
2 // Definição de função-membro da classe BinarySearch.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstdlib> // protótipos para as funções srand e rand
8 using std::rand;
9 using std::srand;
10
11 #include <ctime> // protótipo para a função time
12 using std::time;
13
14 #include <algorithm> // protótipo para a função sort
15 #include "BinarySearch.h" // definição da classe BinarySearch
16
17 // construtor inicializa vetor com ints aleatórios e classifica o vetor
18 BinarySearch::BinarySearch(int vectorSize)
19 {
20 size = (vectorSize > 0 ? vectorSize : 10); // valida vectorSize
21 srand(time(0)); // semeia utilizando a hora atual
22
23 // preenche o vetor com ints aleatórios no intervalo de 10-99
24 for (int i = 0; i < size; i++)
25 data.push_back(10 + rand() % 90); // 10-99
26
27 std::sort(data.begin(), data.end()); // classifica os dados
28 } // fim do construtor BinarySearch
29
30 // realiza uma pesquisa linear nos dados
31 int BinarySearch::binarySearch(int searchElement) const
32 {
33 int low = 0; // extremidade baixa da área de pesquisa
34 int high = size - 1; // extremidade alta da área de pesquisa
35 int middle = (low + high + 1) / 2; // elemento intermediário
36 int location = -1; // valor de retorno; -1 se não localizado
37
38 do // faz um loop para procurar o elemento
39 {
40 // imprime elementos restantes de vetor a ser pesquisado
41 displaySubElements(low, high);
42
43 // gera espaços para alinhamento
44 for (int i = 0; i < middle; i++)
45 cout << " ";

```

**Figura 20.3** Definição de função-membro da classe `BinarySearch`.

(continua)

```

46
47 cout << " * " << endl; // indica o meio atual
48
49 // se o elemento for localizado no meio
50 if (searchElement == data[middle])
51 location = middle; // a localização é o meio atual
52 else if (searchElement < data[middle]) // o meio é muito alto
53 high = middle - 1; // elimina a metade mais alta
54 else // elemento do meio é muito baixo
55 low = middle + 1; // elimina a metade mais baixa
56
57 middle = (low + high + 1) / 2; // recalcula o meio
58 } while ((low <= high) && (location == -1));
59
60 return location; // retorna a localização da chave de pesquisa
61 } // fim da função binarySearch
62
63 // exibe valores no vetor
64 void BinarySearch::displayElements() const
65 {
66 displaySubElements(0, size - 1);
67 } // fim da função displayElements
68
69 // exibe certos valores no vetor
70 void BinarySearch::displaySubElements(int low, int high) const
71 {
72 for (int i = 0; i < low; i++) // gera espaços para alinhamento
73 cout << " ";
74
75 for (int i = low; i <= high; i++) // gera saída dos elementos deixados no vetor
76 cout << data[i] << " ";
77
78 cout << endl;
79 } // fim da função displaySubElements

```

**Figura 20.3** Definição de função-membro da classe BinarySearch.

(continuação)

As linhas 25–41 da Figura 20.4 iteram até que o usuário insira o valor -1. Para todos os outros números que o usuário insere, o programa realiza uma pesquisa binária nos dados para determinar se eles correspondem a um elemento no vetor. A primeira linha da saída desse programa é o vetor de ints, na ordem crescente. Quando o usuário instrui o programa a procurar 38, o programa primeiro testa o elemento intermediário, que é 67 (como indicado por \*). A chave de pesquisa é menor que 67, assim o programa elimina a segunda metade do vetor e testa o elemento do meio a partir da primeira metade do vetor. A chave de pesquisa é igual a 38, então o programa retorna o índice 3.

### *Eficiência da pesquisa binária*

No cenário do pior caso, pesquisar um vetor classificado de 1.023 elementos levará apenas 10 comparações ao utilizar uma pesquisa binária. Dividir repetidamente 1.023 por 2 (porque, depois de cada comparação, somos capazes de eliminar metade do vetor) e arredondar para baixo (porque também removemos o elemento do meio) produz os valores 511, 255, 127, 63, 31, 15, 7, 3, 1 e 0. O número 1.023 ( $2^{10} - 1$ ) é dividido por 2 apenas 10 vezes para obter o valor 0, o que indica que não há mais elementos a testar. Dividir por 2 é equivalente a uma comparação no algoritmo de pesquisa binária. Portanto, um vetor de 1.048.575 ( $2^{20} - 1$ ) elementos exige no máximo 20 comparações para localizar a chave e um vetor de mais de um bilhão de elementos demanda no máximo 30 comparações para localizar a chave. Isso é uma tremenda melhoria no desempenho em relação à pesquisa linear. Para um vetor de um bilhão de elementos representa uma diferença entre uma média de 500 milhões de comparações para a pesquisa linear e no máximo apenas 30 comparações para a pesquisa binária! O número máximo necessário de comparações para a pesquisa binária de qualquer vetor classificado é o expoente da primeira potência de 2 maior que o número de elementos no vetor, que é representado como  $\log_2 n$ . Todos os logaritmos aumentam aproximadamente à mesma taxa, então, na notação O a base pode ser omitida. Isso resulta em uma notação O de  $O(\log n)$  para uma pesquisa binária, que também é conhecida como **tempo de execução logarítmico** e é pronunciada ‘na ordem de  $\log n$ ’ ou mais simplesmente ‘**ordem  $\log n$** ’.



```

1 // Figura 20.4: Fig20_04.cpp
2 // Programa de teste BinarySearch.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6 using std::endl;
7
8 #include "BinarySearch.h" // definição da classe BinarySearch
9
10 int main()
11 {
12 int searchInt; // chave de pesquisa
13 int position; // localização da chave de pesquisa no vetor
14
15 // cria vetor e gera sua saída
16 BinarySearch searchVector (15);
17 searchVector.displayElements();
18
19 // obtém a entrada de usuário
20 cout << "\nPlease enter an integer value (-1 to quit): ";
21 cin >> searchInt; // lê um int do usuário
22 cout << endl;
23
24 // insere repetidamente um inteiro; -1 termina o programa
25 while (searchInt != -1)
26 {
27 // utiliza a pesquisa binária para tentar localizar o inteiro
28 position = searchVector.binarySearch(searchInt);
29
30 // valor de retorno de -1 indica que o inteiro não foi localizado
31 if (position == -1)
32 cout << "The integer " << searchInt << " was not found.\n";
33 else
34 cout << "The integer " << searchInt
35 << " was found in position " << position << ".\n";
36
37 // obtém a entrada de usuário
38 cout << "\n\nPlease enter an integer value (-1 to quit): ";
39 cin >> searchInt; // lê um int do usuário
40 cout << endl;
41 } // fim do while
42
43 return 0;
44 } // fim do main

```

26 31 33 38 47 49 49 67 73 74 82 89 90 91 95

Please enter an integer value (-1 to quit): **38**

26 31 33 38 47 49 49 67 73 74 82 89 90 91 95

\*

26 31 33 38 47 49 49

\*

26 31 33 38 47 49 49 67 73 74 82 89 90 91 95

**Figura 20.4** Programa de teste BinarySearch.

(continua)

```

Please enter an integer value (-1 to quit): 38

26 31 33 38 47 49 49 67 73 74 82 89 90 91 95
 *
26 31 33 38 47 49 49
 *
The integer 38 was found in position 3.

Please enter an integer value (-1 to quit): 91

26 31 33 38 47 49 49 67 73 74 82 89 90 91 95
 *
 73 74 82 89 90 91 95
 *
 90 91 95
 *
The integer 91 was found in position 13.

Please enter an integer value (-1 to quit): 25

26 31 33 38 47 49 49 67 73 74 82 89 90 91 95
 *
26 31 33 38 47 49 49
 *
26 31 33
 *
26
*
The integer 25 was not found.

Please enter an integer value (-1 to quit): -1

```

**Figura 20.4** Programa de teste BinarySearch.

(continuação)

## 20.3 Algoritmos de classificação

Classificar dados (isto é, colocar os dados em alguma ordem particular como crescente ou decrescente) é uma das aplicações mais importantes da computação. Um banco classifica todos os cheques pelo número de conta de modo que possa preparar extratos bancários individuais no final de cada mês. As companhias telefônicas classificam suas listas de assinantes por sobrenome e, depois, pelo primeiro nome para facilitar a localização de números de telefone. Praticamente todas as empresas devem classificar alguns dados e, freqüentemente, volumes maciços deles. Classificar dados é um problema intrigante, que faz uso intensivo do computador e atrai esforços intensos de pesquisa.

Um ponto importante a entender sobre a classificação é que o resultado final — o vetor classificado — será o mesmo independentemente do algoritmo que você utiliza para classificar o vetor. A escolha do algoritmo só afeta o tempo de execução e o uso de memória do programa. Nos capítulos anteriores, introduzimos a classificação por seleção e por inserção — algoritmos simples de implementar, mas ineficientes. A próxima seção examina a eficiência desses dois algoritmos utilizando a notação  $O$ . O último algoritmo — a classificação por intercalação, que introduzimos neste capítulo — é muito mais rápido, porém mais difícil de implementar.

### 20.3.1 Eficiência da classificação por seleção

A classificação por seleção é um algoritmo de classificação fácil de implementar, mas ineficiente. A primeira iteração do algoritmo seleciona o menor elemento no vetor e o permuta pelo primeiro elemento. A segunda iteração seleciona o segundo menor elemento (que é o menor dos elementos restantes) e o permuta pelo segundo elemento. O algoritmo continua até que a última iteração selecione o segundo maior elemento e permuta-o pelo penúltimo elemento, deixando o maior elemento no último índice. Depois da  $i$ -ésima iteração, os menores  $i$  elementos do vetor serão classificados na ordem crescente dos primeiros  $i$  elementos do vetor.

O algoritmo de classificação por seleção itera  $n - 1$  vezes, colocando a cada passagem o menor elemento restante em sua posição classificada. Localizar o menor elemento restante requer  $n - 1$  comparações durante a primeira iteração,  $n - 2$  durante a segunda iteração e, então,  $n - 3, \dots, 3, 2, 1$ . Isso resulta em um total de  $n(n - 1) / 2$  ou  $(n^2 - n) / 2$  comparações. Na notação  $O$ , os menores termos são eliminados e as constantes são ignoradas, deixando um  $O$  final de  $O(n^2)$ .

### 20.3.2 Eficiência da classificação por inserção

A classificação por inserção é outro algoritmo de classificação simples, mas ineficiente. A primeira iteração desse algoritmo seleciona o segundo elemento no vetor e, se for menor que o primeiro elemento, permuta-o pelo primeiro elemento. A segunda iteração examina o terceiro elemento e o insere na posição correta com relação aos dois primeiros elementos de modo que todos os três elementos estejam na ordem. Na  $i$ -ésima iteração desse algoritmo, os primeiros  $i$  elementos no vetor original estarão classificados.

A classificação por inserção itera  $n - 1$  vezes, inserindo um elemento na posição apropriada nos elementos classificados até agora. Para cada iteração, determinar onde inserir o elemento requer comparar o elemento a cada um dos elementos precedentes no vetor. No pior caso, isso exigirá  $n - 1$  comparações. Cada instrução de repetição individual executa em tempo  $O(n)$ . Para determinar a notação  $O$ , as instruções aninhadas querem dizer que você deve multiplicar o número de comparações. Para cada iteração de um loop externo, haverá certo número de iterações do loop interno. Nesse algoritmo, para cada  $O(n)$  iteração do loop externo, há  $O(n)$  iterações do loop interno, resultando em um  $O$  de  $O(n * n)$  ou  $O(n^2)$ .

### 20.3.3 Classificação por intercalação (uma implementação recursiva)

A **classificação por intercalação** é um algoritmo de classificação eficiente, porém conceitualmente mais complexo que a classificação por seleção e a classificação por inserção. O algoritmo de classificação por intercalação classifica um vetor dividindo-o em dois subvetores do mesmo tamanho, classificando cada subvetor e, então, mesclando-os em um vetor maior. Com um número ímpar de elementos, o algoritmo cria os dois subvetores de tal maneira que um deles tenha um elemento a mais que o outro.

A implementação da classificação por intercalação nesse exemplo é recursiva. O caso básico é um vetor com um elemento. Um vetor de um elemento está naturalmente classificado, assim a classificação por intercalação retorna imediatamente quando é chamada com um vetor de um elemento. O passo de recursão divide um vetor de dois ou mais elementos em dois subvetores do mesmo tamanho, classifica recursivamente cada subvetor, e então os mescla em um vetor classificado maior. [Novamente, se houver um número ímpar de elementos, um subvetor é um elemento maior que o outro.]

Suponha que o algoritmo já tenha intercalado os vetores menores para criar os vetores classificados A:

4 10 34 56 77

e B:

5 30 51 52 93

A classificação por intercalação combina esses dois vetores em um vetor classificado maior. O menor elemento em A é 4 (localizado no zero-ésimo índice de A). O menor elemento em B é 5 (localizado no zero-ésimo índice de B). A fim de determinar o menor elemento no maior vetor, o algoritmo compara 4 e 5. O valor em A é menor, assim 4 torna-se o primeiro elemento no vetor intercalado. O algoritmo continua comparando 10 (o segundo elemento em A) com 5 (o primeiro elemento em B). O valor de B é menor, portanto 5 torna-se o segundo elemento no maior vetor. O algoritmo continua comparando 10–30, com 10 tornando-se o terceiro elemento no vetor e assim por diante.

A Figura 20.5 define a classe MergeSort e as linhas 31–34 da Figura 20.6 definem a função sort. A linha 33 chama a função sortSubVector com 0 e size -1 como argumentos. Os argumentos correspondem aos índices inicial e final do vetor a ser classificado, fazendo com que sortSubVector opere no vetor inteiro. A função sortSubVector é definida nas linhas 37–61. A linha 40 testa o caso básico. Se o tamanho do vetor for 1, o vetor já está classificado, então a função simplesmente retorna imediatamente. Se o tamanho do vetor for maior que 1, a função divide o vetor em dois, chama recursivamente a função sortSubVector para classificar os dois subvetores e, então, os intercala. A linha 55 chama recursivamente a função sortSubVector na primeira metade do vetor, e a linha 56 chama recursivamente a função sortSubVector na segunda metade do vetor. Quando essas duas chamadas de função retornam, as duas metades do vetor já foram classificadas. A linha 59 chama a função merge (linhas 64–108) nas duas metades do vetor para combinar os dois vetores classificados em um vetor classificado maior.

As linhas 79–87 na função merge fazem loop até o programa alcançar o fim de um dos dois subvetores. A linha 83 testa qual elemento no início dos vetores é o menor. Se o elemento no vetor esquerdo for menor, a linha 84 o coloca na posição do vetor combinado. Se o elemento no vetor direito for menor, a linha 86 o coloca na posição do vetor combinado. Quando o loop while completar (linha 87), um subvetor inteiro será colocado no vetor combinado, mas o outro subvetor ainda conterá dados. A linha 89 testa se o vetor esquerdo alcançou o fim. Se tiver alcançado, as linhas 91–92 preenchem o vetor combinado com os elementos do vetor direito. Se o vetor esquerdo não tiver alcançado o fim, então o vetor direito deve ter alcançado o fim, e as linhas 96–97 preenchem o vetor combinado com os elementos do vetor esquerdo. Por fim, as linhas 101–102 copiam o vetor combinado para o vetor original. A Figura 20.7 cria e utiliza um objeto MergeSort. A saída desse programa exibe as divisões e intercalações realizadas pela classificação por intercalação, mostrando o progresso da classificação em cada passo do algoritmo.

```

1 // Figura 20.5: MergeSort.h
2 // Classe que cria um vetor preenchido com inteiros aleatórios.
3 // Fornece uma função para classificar o vetor com classificação por intercalação.
4 #include <vector>
5 using std::vector;
6
7 // definição da classe MergeSort
8 class MergeSort
9 {
10 public:
11 MergeSort(int); // construtor inicializa vetor
12 void sort(); // classifica o vetor utilizando classificação por intercalação
13 void displayElements() const; // exibe elementos do vetor
14 private:
15 int size; // tamanho do vetor
16 vector< int > data; // vetor de ints
17 void sortSubVector(int, int); // classifica o subvetor
18 void merge(int, int, int, int); // mescla os dois vetores classificados
19 void displaySubVector(int, int) const; // exibe o subvetor
20 }; // fim da classe MergeSort

```

**Figura 20.5** Definição da classe MergeSort.

```

1 // Figura 20.6: MergeSort.cpp
2 // Definição de função-membro da classe MergeSort.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <vector>
8 using std::vector;
9
10 #include <cstdlib> // protótipos para as funções srand e rand
11 using std::rand;
12 using std::srand;
13
14 #include <ctime> // protótipo para a função time
15 using std::time;
16
17 #include "MergeSort.h" // definição da classe MergeSort
18
19 // construtor preenche vetor com inteiros aleatórios
20 MergeSort::MergeSort(int vectorSize)
21 {
22 size = (vectorSize > 0 ? vectorSize : 10); // valida vectorSize
23 srand(time(0)); // semeia gerador de números aleatórios utilizando a hora atual
24
25 // preenche o vetor com ints aleatórios no intervalo de 10-99
26 for (int i = 0; i < size; i++)
27 data.push_back(10 + rand() % 90);
28 } // fim do construtor de MergeSort
29
30 // divide o vetor, classifica os subvetores e intercala os subvetores no vetor classificado
31 void MergeSort::sort()

```

**Figura 20.6** Definição de função-membro da classe MergeSort.

(continua)

```

32 {
33 sortSubVector(0, size - 1); // classifica recursivamente o vetor inteiro
34 } // fim da função sort
35
36 // função recursiva para classificar subvetores
37 void MergeSort::sortSubVector(int low, int high)
38 {
39 // caso básico de teste; tamanho do vetor é igual a 1
40 if ((high - low) >= 1) // se não for o caso básico
41 {
42 int middle1 = (low + high) / 2; // calcula o meio do vetor
43 int middle2 = middle1 + 1; // calcula o próximo elemento
44
45 // gera saída do passo de divisão
46 cout << "split: ";
47 displaySubVector(low, high);
48 cout << endl << " ";
49 displaySubVector(low, middle1);
50 cout << endl << " ";
51 displaySubVector(middle2, high);
52 cout << endl << endl;
53
54 // divide o vetor pela metade; classifica cada metade (chamadas recursivas)
55 sortSubVector(low, middle1); // primeira metade do vetor
56 sortSubVector(middle2, high); // segunda metade do vetor
57
58 // intercala 2 vetores classificados após as chamadas de divisão retornarem
59 merge(low, middle1, middle2, high);
60 } // fim do if
61 } // fim da função sortSubVector
62
63 // intercala dois subvetores classificados em um subvetor classificado
64 void MergeSort::merge(int left, int middle1, int middle2, int right)
65 {
66 int leftIndex = left; // índice no subvetor esquerdo
67 int rightIndex = middle2; // índice no subvetor direito
68 int combinedIndex = left; // índice no vetor de trabalho temporário
69 vector< int > combined(size); // vetor de trabalho
70
71 // gera saída dos dois subvetores antes da intercalação
72 cout << "merge: ";
73 displaySubVector(left, middle1);
74 cout << endl << " ";
75 displaySubVector(middle2, right);
76 cout << endl;
77
78 // intercala vetores até alcançar o fim de um deles
79 while (leftIndex <= middle1 && rightIndex <= right)
80 {
81 // coloca o menor dos dois elementos atuais no resultado
82 // e se move para o próximo espaço no vetor
83 if (data[leftIndex] <= data[rightIndex])
84 combined[combinedIndex++] = data[leftIndex++];
85 else
86 combined[combinedIndex++] = data[rightIndex++];
87 } // fim do while
88

```

Figura 20.6 Definição de função-membro da classe MergeSort.

(continua)

```

89 if (leftIndex == middle2) // se no fim do vetor esquerdo
90 {
91 while (rightIndex <= right) // copia o restante do vetor direito
92 combined[combinedIndex++] = data[rightIndex++];
93 } // fim do if
94 else // no fim do vetor direito
95 {
96 while (leftIndex <= middle1) // copia o restante do vetor esquerdo
97 combined[combinedIndex++] = data[leftIndex++];
98 } // fim do else
99
100 // copia valores de volta para o vetor original
101 for (int i = left; i <= right; i++)
102 data[i] = combined[i];
103
104 // gera saída do vetor intercalado
105 cout << " ";
106 displaySubVector(left, right);
107 cout << endl << endl;
108 } // fim da função merge
109
110 // exibe elementos no vetor
111 void MergeSort::displayElements() const
112 {
113 displaySubVector(0, size - 1);
114 } // fim da função displayElements
115
116 // exibe certos valores no vetor
117 void MergeSort::displaySubVector(int low, int high) const
118 {
119 // gera espaços para alinhamento
120 for (int i = 0; i < low; i++)
121 cout << " ";
122
123 // gera saída dos elementos deixados no vetor
124 for (int i = low; i <= high; i++)
125 cout << " " << data[i];
126 } // fim da função displaySubVector

```

**Figura 20.6** Definição de função-membro da classe MergeSort.

(continuação)

### *Eficiência da classificação por intercalação*

A classificação por intercalação é um algoritmo muito mais eficiente que a classificação por inserção ou por seleção (embora possa ser difícil de acreditar nisso ao examinar a complexa Figura 20.7). Considere a primeira chamada (não recursiva) à função `sortSubVector`. Isso resulta em duas chamadas recursivas à função `sortSubVector` em que cada subvetor apresenta aproximadamente a metade do tamanho do vetor original e uma única chamada à função `merge`. Essa chamada à função `merge` exige, no pior dos casos,  $n - 1$  comparações para preencher o vetor original, que é  $O(n)$ . (Lembre-se de que cada elemento no vetor é escolhido comparando um elemento a partir de cada um dos subvetores.) As duas chamadas à função `sortSubVector` resultam em mais quatro chamadas recursivas à função `sortSubVector`, cada uma com um subvetor de aproximadamente um quarto do tamanho do vetor original, junto com duas chamadas à função `merge`. Cada uma dessas duas chamadas à função `merge` exige, no pior dos casos,  $n/2 - 1$  comparações, para um número total de comparações de  $O(n)$ . Esse processo continua, e cada chamada a `sortSubVector` gera duas chamadas adicionais a `sortSubVector` e uma chamada a `merge`, até que o algoritmo tenha dividido o vetor em subvetores de um elemento. Em cada nível,  $O(n)$  comparações são exigidas para intercalar os subvetores. Cada nível divide o tamanho dos vetores pela metade, portanto dobrar o tamanho do vetor exige mais um nível. Quadruplicar o tamanho do vetor exige mais dois níveis. Esse padrão é logarítmico e resulta em  $\log_2 n$  níveis. Isso resulta em uma eficiência total de  $O(n \log n)$ . A Figura 20.8 resume boa parte dos algoritmos de pesquisa e classificação abrangidos neste livro e lista a notação  $O$  para cada um deles. A Figura 20.9 lista os valores de  $O$  que abordamos neste capítulo junto com alguns valores para  $n$  a fim de destacar as diferenças nas taxas de crescimento.

```

1 // Figura 20.7: Fig20_07.cpp
2 // Programa de teste MergeSort.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "MergeSort.h" // definição da classe MergeSort
8
9 int main()
10 {
11 // cria o objeto para executar a classificação por intercalação
12 MergeSort sortVector(10);
13
14 cout << "Unsorted vector:" << endl;
15 sortVector.displayElements(); // imprime o vetor não classificado
16 cout << endl << endl;
17
18 sortVector.sort(); // classifica o vetor
19
20 cout << "Sorted vector:" << endl;
21 sortVector.displayElements(); // imprime o vetor classificado
22 cout << endl;
23 return 0;
24 } // fim do main

```

Unsorted vector:

30 47 22 67 79 18 60 78 26 54

split:    30 47 22 67 79 18 60 78 26 54  
           30 47 22 67 79  
                         18 60 78 26 54

split:    30 47 22 67 79  
           30 47 22  
                         67 79

split:    30 47 22  
           30 47  
                         22

split:    30 47  
           30  
                         47

merge:    30  
           47  
           30 47

merge:    30 47  
                         22  
           22 30 47

split:               67 79  
                         67  
                                 79

**Figura 20.7** Programa de teste MergeSort.

(continua)



```

merge: 67
 79
 67 79

merge: 22 30 47
 67 79
 22 30 47 67 79

split: 18 60 78 26 54
 18 60 78
 26 54

split: 18 60 78
 18 60
 78

split: 18 60
 18
 60

merge: 18
 60
 18 60

merge: 18 60
 78
 18 60 78

split: 26 54
 26
 54

merge: 26
 54
 26 54

merge: 18 60 78
 26 54
 18 26 54 60 78

merge: 22 30 47 67 79
 18 26 54 60 78
 18 22 26 30 47 54 60 67 78 79

Sorted vector:
18 22 26 30 47 54 60 67 78 79

```

**Figura 20.7** Programa de teste MergeSort.

(continuação)

## 20.4 Síntese

Este capítulo discutiu a pesquisa e a classificação de dados. Discutimos o algoritmo de pesquisa binária — um algoritmo de pesquisa mais rápido, mas mais complexo do que a pesquisa linear (Seção 7.7). O algoritmo de pesquisa binária só funcionará em um vetor classificado, mas cada iteração de pesquisa binária elimina metade dos elementos do vetor. Você também aprendeu o algoritmo de classificação por intercalação, que é um algoritmo de classificação mais eficiente que qualquer classificação por inserção (Seção 7.8) ou por seleção (Seção 8.6). Também foi feita uma introdução à notação  $O$  que ajuda a expressar a eficiência de um algoritmo. A notação  $O$  mede o tempo de execução de pior caso de um algoritmo. O valor  $O$  de um algoritmo é útil para comparar algoritmos e escolher o mais eficiente. No próximo capítulo, você aprenderá as estruturas de dados dinâmicas que podem aumentar ou diminuir em tempo de execução.

Algoritmo	Posição	Notação O
<i>Algoritmos de pesquisa</i>		
Pesquisa linear	Seção 7.7	$O(n)$
Pesquisa binária	Seção 20.2.2	$O(\log n)$
Pesquisa linear recursiva	Exercício 20.8	$O(n)$
Pesquisa binária recursiva	Exercício 20.9	$O(\log n)$
<i>Algoritmos de classificação</i>		
Classificação por inserção	Seção 7.8	$O(n^2)$
Classificação por seleção	Seção 8.6	$O(n^2)$
Classificação por intercalação	Seção 20.3.3	$O(n \log n)$
Classificação por borbulhamento ( <i>bubble sort</i> )	Exercícios 16.3 e 16.4	$O(n^2)$
Classificação rápida	Exercício 20.10	Pior caso: $O(n^2)$ Caso médio: $O(n \log n)$

**Figura 20.8** Algoritmos de pesquisa e classificação com valores na notação O.

$n$	Valor decimal aproximado	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$
$2^{10}$	1.000	10	$2^{10}$	$10 \cdot 2^{10}$	$2^{20}$
$2^{20}$	1.000.000	20	$2^{20}$	$20 \cdot 2^{20}$	$2^{40}$
$2^{30}$	1.000.000.000	30	$2^{30}$	$30 \cdot 2^{30}$	$2^{60}$

**Figura 20.9** O número aproximado de comparações para notações O comuns.

## Resumo

- Pesquisar dados envolve determinar se uma chave de pesquisa está presente nos dados e, se estiver, encontrar sua localização.
- Classificar envolve organizar os dados em uma ordem.
- O algoritmo de pesquisa linear pesquisa cada elemento no vetor sequencialmente até encontrar o elemento correto. Se o elemento não estiver no vetor, o algoritmo testa cada elemento no vetor e, quando o fim do vetor é alcançado, informa o usuário de que o elemento não está presente. Se o elemento estiver no vetor, a pesquisa linear testa cada elemento até localizar o correto.
- Uma diferença importante entre algoritmos de pesquisa é o esforço que eles exigem a fim de retornar um resultado.
- Uma maneira de descrever a eficiência de um algoritmo é com a notação ( $O$ ), que indica a dificuldade que um algoritmo pode ter para resolver um problema.
- Para algoritmos de pesquisa e classificação, a notação O descreve como a quantidade de esforço de um algoritmo particular varia dependendo da quantidade de elementos nos dados.
- Diz-se que um algoritmo que é  $O(1)$  tem um tempo de execução constante. Isso não significa que o algoritmo requer somente uma comparação. Significa apenas que o número de comparações não aumenta à medida que o tamanho do vetor aumenta.
- Diz-se que um algoritmo  $O(n)$  tem um tempo de execução linear.
- A notação O é projetada para destacar fatores dominantes e ignorar termos sem importância com valores de  $n$  altos.
- A notação O se preocupa com a taxa de crescimento dos tempos de execução do algoritmo, portanto constantes são ignoradas.
- O algoritmo de pesquisa linear executa em tempo  $O(n)$ .
- O pior caso na pesquisa linear é que cada elemento deve ser verificado para determinar se o elemento de pesquisa existe. Isso ocorre se a chave de pesquisa for o último elemento no vetor ou não estiver presente.

- O algoritmo de pesquisa binária é mais eficiente que o algoritmo de pesquisa linear, mas exige que primeiro o vetor seja classificado. Isso só é vantajoso quando o vetor, uma vez classificado, for pesquisado várias vezes — ou quando o aplicativo de pesquisa tiver rigorosos requisitos de desempenho.
- A primeira iteração da pesquisa binária testa o elemento do meio no vetor. Se este for a chave de pesquisa, o algoritmo retornará sua localização. Se a chave de pesquisa for menor que o elemento do meio, a pesquisa binária continuará com a primeira metade do vetor. Se a chave de pesquisa for maior que o elemento do meio, a pesquisa binária continuará com a segunda metade do vetor. Cada iteração da pesquisa binária testa o valor do meio do vetor restante e, se o elemento não for localizado, elimina metade dos elementos restantes.
- A pesquisa binária é mais eficiente que a linear porque, em cada comparação, ela elimina da consideração metade dos elementos no vetor.
- A pesquisa binária executa em tempo  $O(\log n)$ , pois cada passo elimina da consideração metade dos elementos remanescentes.
- Se o tamanho do vetor for dobrado, a pesquisa binária exigirá somente uma comparação extra para completar com sucesso.
- A classificação por seleção é um algoritmo de classificação simples, mas ineficiente.
- A primeira iteração da classificação por seleção seleciona o menor elemento no vetor e o permuta pelo primeiro elemento. A segunda iteração da classificação por seleção seleciona o segundo menor elemento (que é o menor elemento restante) e o permuta pelo segundo elemento. A classificação por seleção continua até que a última iteração selecione o segundo maior elemento e o permuta pelo penúltimo índice, deixando o maior elemento no último índice. Na  $i$ -ésima iteração da classificação por seleção, os menores  $i$  elementos do vetor inteiro são classificados nos primeiros  $i$  índices.
- O algoritmo de classificação por seleção executa no tempo  $O(n^2)$ .
- A primeira iteração da classificação por inserção seleciona o segundo elemento no vetor e, se for menor que o primeiro elemento, permuta-o pelo primeiro elemento. A segunda iteração de classificação por inserção examina o terceiro elemento e o insere na posição correta com relação aos dois primeiros elementos. Depois da  $i$ -ésima iteração da classificação por inserção, os primeiros  $i$  elementos no vetor original são classificados. Somente  $n - 1$  iterações são necessárias.
- O algoritmo de classificação por inserção executa no tempo  $O(n^2)$ .
- A classificação por intercalação é um algoritmo de classificação mais rápido, porém mais complexo de implementar, que a classificação por seleção e a classificação por inserção.
- O algoritmo de classificação por intercalação classifica um vetor dividindo o vetor em dois subvetores do mesmo tamanho, classificando cada subvetor e intercalando-os em um vetor maior.
- O caso básico da classificação por intercalação é um vetor com um elemento. O vetor de um elemento já está classificado, assim a classificação por intercalação retorna imediatamente quando é chamada com um vetor de um elemento. A parte da classificação por intercalação seleciona dois vetores classificados (estes poderiam ser vetores de um elemento) e combina-os em um vetor classificado maior.
- A classificação por intercalação realiza a intercalação examinando o primeiro elemento em cada vetor, que também é o menor elemento no vetor. A classificação por intercalação seleciona o menor destes e coloca-os no primeiro elemento do maior vetor classificado. Se ainda houver elementos no subvetor, a classificação por intercalação examina o segundo elemento nesse subvetor (que agora é o menor elemento remanescente) e o compara ao primeiro elemento no outro subvetor. A classificação por intercalação continua esse processo até que o maior vetor seja preenchido.
- No pior caso, a primeira chamada à classificação por intercalação tem de fazer  $O(n)$  comparações para preencher os  $n$  slots no vetor final.
- A parte da intercalação do algoritmo de classificação por intercalação é realizada em dois subvetores, cada um com aproximadamente o tamanho  $n/2$ . Criar cada um desses subvetores exige  $n/2 - 1$  comparações para cada subvetor ou o total de  $O(n)$  comparações. Esse padrão continua à medida que cada nível constrói duas vezes o número de vetores, mas cada um tem a metade do tamanho do vetor anterior.
- Semelhantemente à pesquisa binária, essa divisão em metades resulta em  $\log n$  níveis, cada nível exigindo  $O(n)$  comparações, para uma eficiência total de  $O(n \log n)$ .

## Terminologia

chave de classificação	$O(1)$	pesquisa binária
chave de pesquisa	$O(\log n)$	pesquisando dados
classificação por intercalação	$O(n \log n)$	tempo de execução constante
classificando dados	$O(n)$	tempo de execução linear
divisão do vetor na classificação por intercalação	$O(n^2)$	tempo de execução logarítmico
função sort da Standard Library	ordem 1	tempo de execução no pior cenário de um algoritmo
intercalar (mesclar) dois vetores	ordem do quadrado de $n$	tempo de execução quadrático
iterador de acesso aleatório	ordem $\log n$	
notação $O$	ordem $n$	

## Exercícios de revisão

- 20.1** Preencha as lacunas em cada uma das seguintes sentenças:
- Um programa de classificação por seleção demoraria aproximadamente \_\_\_\_\_ vezes mais para ser executado em um vetor de 128 elementos do que em um vetor de 32 elementos.
  - A eficiência da classificação por intercalação é \_\_\_\_\_.
- 20.2** Qual aspecto-chave da pesquisa binária e da classificação por intercalação é responsável pela parte logarítmica das suas respectivas notações  $O$ s?
- 20.3** Em que sentido a classificação por inserção é superior à classificação por intercalação? Em que sentido a classificação por intercalação é superior à classificação por inserção?
- 20.4** No texto, dizemos que, depois que a classificação por intercalação divide o vetor em dois subvetores, ela então classifica esses dois subvetores e os intercala. Por que alguém ficaria intrigado com a nossa afirmação de que “ela então classifica esses dois subvetores”?

## Respostas dos exercícios de revisão

- 20.1** a) 16, porque um algoritmo  $O(n^2)$  demora 16 vezes mais para classificar quatro vezes o mesmo número de informações. b)  $O(n \log n)$ .
- 20.2** Esses dois algoritmos incorporam ‘a divisão em metades’— reduzindo algo à metade, de alguma maneira. A pesquisa binária elimina uma metade do vetor depois de cada comparação. A classificação por intercalação divide o vetor pela metade toda vez que é chamada.
- 20.3** A classificação por inserção é mais fácil de entender e implementar do que a classificação por intercalação. A classificação por intercalação é muito mais eficiente ( $O(n \log n)$ ) do que a classificação por inserção ( $O(n^2)$ ).
- 20.4** Em certo sentido, ela na verdade não classifica esses dois subvetores. Simplesmente continua a dividir o vetor original pela metade até que ele forneça um subvetor de um elemento, que está naturalmente classificado. Então ele cria os dois subvetores originais mesclando esses vetores de um elemento para formar subvetores maiores, que são então mesclados e assim por diante.

## Exercícios

[Nota: A maioria dos exercícios mostrados neste capítulo é duplicata dos exercícios dos capítulos 7–8. Incluímos os exercícios novamente aqui, por conveniência, para que os leitores estudem a pesquisa e classificação neste capítulo.]

- 20.5** (*Classificação por borbulhamento [bubble sort]*) Implemente a classificação por borbulhamento — outra técnica simples mas ineficiente de classificação. É chamada classificação por borbulhamento ou classificação por afundamento porque os menores valores gradualmente ‘borbulham’ para a parte superior do vetor (isto é, na direção do primeiro elemento) como bolhas de ar que emergem na superfície, enquanto os maiores valores afundam para a parte inferior (final) do vetor. A técnica utiliza loops aninhados para fazer várias passagens pelo vetor. Cada passagem compara pares sucessivos de elementos. Se um par estiver na ordem crescente (ou os valores forem iguais), a classificação por borbulhamento deixa os valores como estão. Se um par estiver na ordem decrescente, a classificação por borbulhamento permuta seus valores no vetor.

A primeira passagem compara os dois primeiros elementos do vetor e os permuta se necessário. Ela então compara o segundo e terceiro elementos no vetor. O final dessa passagem compara os dois últimos elementos no vetor e os permuta se necessário. Depois de uma passagem, o maior elemento estará no último índice. Depois de duas passagens, os dois maiores elementos estarão nos dois últimos índices. Explique por que a classificação por borbulhamento é um algoritmo  $O(n^2)$ .

- 20.6** (*Classificação por borbulhamento aprimorada*) Faça as modificações simples a seguir para melhorar o desempenho da classificação por borbulhamento que você desenvolveu no Exercício 20.5:
- Depois da primeira passagem, é garantido que o número maior está no elemento de número mais alto do vetor; após a segunda passagem, os dois números mais altos estão ‘no lugar’; e assim por diante. Em vez de fazer nove comparações (para um vetor de 10 elementos) em cada passagem, modifique a classificação por borbulhamento para fazer apenas as oito comparações necessárias na segunda passagem, sete na terceira e assim por diante.
  - Os dados no vetor já podem estar na ordem adequada ou próxima da ordem adequada, então por que fazer nove passagens (de um vetor de 10 elementos) se menos serão suficientes? Modifique a classificação para verificar no fim de cada passagem se alguma permuta foi feita. Se nenhuma permuta tiver sido feita, os dados já devem estar na ordem apropriada, então o programa deve terminar. Se permutas foram feitas, pelo menos mais uma passagem é necessária.
- 20.7** (*Bucket sort*) Uma classificação do tipo *bucket sort* inicia com um vetor unidimensional de inteiros positivos a ser classificado e um vetor bidimensional de inteiros com linhas indexadas de 0–9 e colunas indexadas de 0 a  $n - 1$ , onde  $n$  é o número dos valores a serem classificados. Cada linha do vetor bidimensional é chamada de *bucket*. Escreva uma classe chamada `BucketSort` que contém uma função chamada `sort` que opera desta maneira:
- Coloque cada valor do vetor unidimensional em uma linha do vetor de bucket, com base no dígito das ‘unidades’ (mais à direita) do valor. Por exemplo, 97 é colocado na linha 7, 3 é colocado na linha 3 e 100 é colocado na linha 0. Esse procedimento é chamado de *passagem de distribuição*.

- b) Realize um loop pelo vetor de bucket linha por linha e copie os valores de volta para o vetor original. Esse procedimento é chamado *passagem de coleta*. A nova ordem dos valores precedentes no vetor unidimensional é 100, 3 e 97.
- c) Repita esse processo para a posição de cada dígito subsequente (dezenas, centenas, milhares etc.).

Na segunda passagem (dígitos das dezenas), 100 é colocado na linha 0, 3 é colocado na linha 0 (porque 3 não tem nenhum dígito de dezena) e 97 é colocado na linha 9. Depois da passagem de coleta, a ordem dos valores no vetor unidimensional é 100, 3 e 97. Na terceira passagem (dígitos das centenas), 100 é colocado na linha 1, 3 é colocado na linha 0 e 97 é colocado na linha 0 (depois do 3). Depois dessa última passagem de coleta, o vetor original está na ordem classificada.

Observe que o vetor bidimensional dos buckets tem 10 vezes o comprimento do vetor de inteiros sendo classificado. Essa técnica de classificação fornece melhor desempenho do que uma classificação por borbulhamento, mas exige muito mais memória — a classificação por borbulhamento exige espaço para somente um elemento adicional de dados. Essa comparação é um exemplo da relação de troca espaço-tempo: a *bucket sort* utiliza mais memória que a classificação por borbulhamento, mas seu desempenho é melhor. Essa versão da *bucket sort* requer cópia de todos os dados de volta para o vetor original a cada passagem. Outra possibilidade é criar um segundo vetor de bucket bidimensional e permutar os dados repetidamente entre os dois vetores de bucket.

**20.8** (*Pesquisa linear recursiva*) Modifique o Exercício 7.33 para utilizar a função recursiva `recursiveLinearSearch` a fim de realizar uma pesquisa linear no vetor. A função deve receber a chave de pesquisa e o índice inicial como argumentos. Se a chave de pesquisa for encontrada, seu índice no vetor é retornado; caso contrário, `-1` é retornado. Cada chamada à função recursiva deve verificar um índice no vetor.

**20.9** (*Pesquisa binária recursiva*) Modifique a Figura 20.3 para utilizar a função recursiva `recursiveBinarySearch` a fim de realizar uma pesquisa binária do vetor. A função deve receber a chave de pesquisa, o índice inicial e o índice final como argumentos. Se a chave de pesquisa for encontrada, seu índice no vetor é retornado. Se a chave de pesquisa não for encontrada, é retornado `-1`.

**20.10** (*Quicksort*) A técnica de classificação recursiva chamada *quicksort* utiliza o seguinte algoritmo básico para um vetor unidimensional de valores:

- a) *Passo de partição*: Selecione o primeiro elemento do vetor não classificado e determine sua localização final no vetor classificado (isto é, todos os valores à esquerda do elemento no vetor são menores que o elemento, e todos os valores à direita do elemento no vetor são maiores que o elemento — mostramos como fazer isso a seguir). Agora temos um elemento em sua posição adequada e dois subvetores não classificados.
- b) *Passo de recursão*: Realize o Passo 1 em cada subvetor não classificado. Toda vez que o Passo 1 for realizado em um subvetor, outro elemento é colocado em sua posição final no vetor classificado e dois subvetores não classificados são criados. Quando um subvetor consiste em apenas um elemento, esse elemento está na sua localização final (porque o vetor de um elemento já está classificado). O algoritmo básico parece suficientemente simples, mas como determinamos a posição final do primeiro elemento de cada subvetor? Como um exemplo, considere o seguinte conjunto de valores (o elemento em negrito é o elemento de particionamento — ele será colocado em sua localização final no vetor classificado):

**37** 2 6 4 89 8 10 12 68 45

Iniciando a partir do elemento mais à direita do vetor, compare cada elemento com **37** até um elemento menor que **37** ser encontrado, então permuta 37 e esse elemento. O primeiro elemento menor que **37** é 12, então **37** e 12 são permutados. O novo vetor é

12 2 6 4 89 8 10 **37** 68 45

O elemento 12 está em itálico para indicar que acabou ser permutado com **37**.

Iniciando a partir da esquerda do vetor, mas começando com o elemento depois de 12, compare cada elemento com **37** até um elemento maior que **37** ser encontrado, então permuta **37** e esse elemento. O primeiro elemento maior que **37** é 89, então **37** e 89 são permutados. O novo vetor é

12 2 6 4 **37** 8 10 89 68 45

Iniciando da direita, mas começando com o elemento antes de 89, compare cada elemento com **37** até um elemento menor que **37** ser encontrado — então permuta **37** e esse elemento. O primeiro elemento menor que **37** é 10, então **37** e 10 são permutados. O novo vetor é

12 2 6 4 10 8 **37** 89 68 45

Iniciando da esquerda, mas começando com o elemento depois de 10, compare cada elemento para **37** até um elemento maior que **37** ser encontrado — então permuta **37** e esse elemento. Não há mais elementos maiores que **37**; então, quando comparamos **37** com ele mesmo, sabemos que **37** foi colocado na sua localização final do vetor classificado. Cada valor à esquerda do **37** é menor que ele e cada valor à direita do **37** é maior que ele.

Uma vez que a partição foi aplicada no vetor anterior, há dois subvetores não classificados. O subvetor com valores menores que 37 contém 12, 2, 6, 4, 10 e 8. O subvetor com valores maiores que 37 contém 89, 68 e 45. A classificação continua recursivamente com ambos os subvetores sendo particionados da mesma maneira que o vetor original.

Com base na discussão precedente, escreva a função recursiva `quickSortHelper` para classificar um vetor unidimensional de inteiros. A função deve receber como argumentos um índice inicial e um índice final no vetor original sendo classificado.



*Muito do que eu prendi, não  
pude libertar; muito do que  
libertei voltou para mim.*  
Lee Wilson Dodd

*'Não dá pra ir mais rápido?','  
disse a enchova para o caracol.  
'Há um delfim atrás de mim, e  
ele está me empurrando.'*  
Lewis Carroll

*Há sempre espaço no ponto  
mais alto.*  
Daniel Webster

*Prossiga — continue andando.*  
Thomas Morton

*Virarei uma nova folha.*  
Miguel de Cervantes

## Estruturas de dados

### OBJETIVOS

Neste capítulo, você aprenderá:

- A formar estruturas de dados encadeadas utilizando ponteiros, classes auto-referenciais e recursão.
- Como criar e manipular estruturas de dados dinâmicas, como listas vinculadas, filas, pilhas e árvores binárias.
- A utilizar árvores de pesquisa binária para pesquisa e classificação de alta velocidade.
- Como são várias aplicações importantes de estruturas de dados vinculadas.
- Como criar estruturas de dados reutilizáveis com templates de classe, herança e composição.

- 21.1 Introdução
- 21.2 Classes auto-referenciais
- 21.3 Alocação de memória e estruturas de dados dinâmicas
- 21.4 Listas vinculadas
- 21.5 Pilhas
- 21.6 Filas
- 21.7 Árvores
- 21.8 Síntese

Resumo | Terminologia | Exercícios de revisão | Respostas dos exercícios de revisão |  
Exercícios | Seção especial: construindo seu próprio compilador

## 21.1 Introdução

Estudamos as **estruturas de dados** de tamanho fixo como arrays unidimensionais, arrays bidimensionais e structs. Este capítulo introduz as **estruturas de dados dinâmicas** que crescem e encolhem durante a execução. As **listas vinculadas** são coleções de itens de dados ‘vinculados em uma cadeia’ — as inserções e as exclusões são feitas em qualquer lugar de uma lista vinculada. As **pilhas** são importantes em compiladores e sistemas operacionais: as inserções e exclusões são feitas somente em uma extremidade de uma pilha — sua **parte superior**. **Filas** (*queues*) representam seqüências de espera; as inserções são feitas na parte de trás (também referida como **cauda**) de uma fila e as remoções são feitas a partir da parte da frente (também referida como **cabeca**) de uma fila. As **árvores binárias** facilitam a pesquisa e a classificação de dados em alta velocidade, a eliminação eficiente de itens de dados duplicados, a representação de diretórios de sistema de arquivos e a compilação de expressões em linguagem de máquina. Essas estruturas de dados têm muitas outras aplicações interessantes.

Discutimos várias estruturas de dados populares e importantes e implementamos programas que as criam e as manipulam. Utilizamos classes, templates de classe, herança e composição para criar e empacotar essas estruturas de dados para facilitar o reuso e a manutenção.

Estudar este capítulo é uma sólida preparação para o Capítulo 23, “Standard Template Library (STL)”. A STL é uma parte importante da Standard Library C++. A STL fornece contêineres, iteradores para percorrer esses contêineres e algoritmos para processar os elementos desses contêineres. Você verá que a STL empacotou cada uma das estruturas de dados que discutimos neste capítulo em classes de template. O código STL é cuidadosamente escrito para ser portátil, eficiente e extensível. Uma vez que entender os princípios e a construção de estruturas de dados da maneira apresentada neste capítulo, você será capaz de fazer o melhor uso de estruturas de dados pré-empacotadas, iteradores e algoritmos na STL, um conjunto de primeira classe de componentes reutilizáveis.

Os exemplos do capítulo são programas práticos que você será capaz de utilizar em cursos mais avançados e em aplicações da indústria. Os programas empregam extensa manipulação de ponteiros. Os exercícios incluem uma rica coleção de aplicativos úteis.

Encorajamos você a tentar o projeto principal descrito na seção especial “Construindo seu próprio compilador”. Você vem utilizando um compilador C++ para converter programas em linguagem de máquina a fim de poder executar esses programas no computador. Nesse projeto, você realmente construirá seu próprio compilador. Ele lerá um arquivo de instruções escrito em uma linguagem simples, mas poderosa e de alto nível, semelhante às primeiras versões da popular linguagem BASIC. Seu compilador traduzirá essas instruções em um arquivo de instruções de Simpletron Machine Language (SML) — SML é a linguagem que você aprendeu na seção especial do Capítulo 8, “Construindo seu próprio computador”. Seu programa Simpletron Simulator então executará o programa SML produzido por seu compilador! A implementação desse projeto utilizando uma abordagem orientada a objetos lhe fornecerá uma maravilhosa oportunidade de praticar grande parte do que você aprendeu neste livro. A seção especial orienta você cuidadosamente nas especificações da linguagem de alto nível e descreve os algoritmos que você precisará para converter cada tipo de instrução de linguagem de alto nível em instruções de linguagem de máquina. Se gosta de desafios, você pode tentar os muitos aprimoramentos no compilador e no Simpletron Simulator sugeridos nos exercícios deste capítulo.

## 21.2 Classes auto-referenciais

Uma **classe auto-referencial** contém um membro ponteiro que aponta para um objeto de classe do mesmo tipo de classe. Por exemplo, a definição

```
class Node
{
public:
 Node(int); // construtor
 void setData(int); // configura membro de dados
 int getData() const; // obtém membro de dados
 void setNextPtr(Node *); // configura ponteiro como próximo Node
```



```

Node *getNextPtr() const; // obtém ponteiro para próximo Node
private:
 int data; // dados armazenados neste Node
 Node *nextPtr; // ponteiro para outro objeto do mesmo tipo
}; // fim da classe Node

```

define um tipo, `Node`. O tipo `Node` tem dois membros de dados `private` — o membro do tipo inteiro `data` e membro ponteiro `nextPtr`. O membro `nextPtr` aponta para um objeto do tipo `Node` — um objeto do mesmo tipo que aquele sendo declarado aqui, daí o termo ‘classe auto-referencial’. O membro `nextPtr` é referido como um **vínculo** (ou *link*) — isto é, `nextPtr` pode ‘vincular’ um objeto de tipo `Node` a outro objeto do mesmo tipo. O tipo `Node` também tem cinco funções-membro — um construtor que recebe um inteiro para inicializar o membro `data`, uma função `setData` para configurar o valor do membro `data`, uma função `getData` para retornar o valor do membro `data`, uma função `setNextPtr` para configurar o valor do membro `nextPtr` e uma função `getNextPtr` para retornar o valor do membro `nextPtr`.

Os objetos de classe auto-referencial podem ser vinculados para formar estruturas de dados úteis como listas, filas, pilhas e árvores. A Figura 21.1 ilustra dois objetos de classe auto-referencial vinculados entre si para formar uma lista. Observe que uma barra — representando um ponteiro nulo (0) — é colocada no membro de vínculo do segundo objeto de classe auto-referencial para indicar que o vínculo não aponta para outro objeto. A barra serve apenas para propósitos de ilustração; ela não corresponde ao caractere de barra invertida em C++. Um ponteiro nulo normalmente indica o fim de uma estrutura de dados, assim como o caractere nulo (‘\0’) indica o fim de uma string.



### Erro comum de programação 21.1

*Não configurar o vínculo no último nó de uma estrutura de dados vinculada como nulo (0) é um erro de lógica (possivelmente fatal).*

## 21.3 Alocação de memória e estruturas de dados dinâmicas

A criação e a manutenção de estruturas de dados dinâmicas requerem alocação dinâmica de memória, o que permite a um programa obter mais memória em tempo de execução para armazenar novos nós. Quando o programa não precisa mais dessa memória, esta pode ser liberada a fim de poder ser reutilizada para alocar outros objetos no futuro. O limite para alocação dinâmica de memória pode ser tão grande quanto a quantidade de memória física disponível no computador ou a quantidade de memória virtual disponível em um sistema de memória virtual. Frequentemente, os limites são muito menores, porque a memória disponível deve ser compartilhada entre muitos programas.

O operador `new` aceita como um argumento o tipo do objeto dinamicamente sendo alocado e retorna um ponteiro para um objeto desse tipo. Por exemplo, a instrução

```
Node *newPtr = new Node(10); // cria Node com o valor de 10
```

aloca os bytes `sizeof( Node )`, executa o construtor `Node` e atribui o endereço do novo objeto `Node` a `newPtr`. Se nenhuma memória estiver disponível, `new` lança uma exceção `bad_alloc`. O valor 10 é passado para o construtor `Node`, que inicializa o membro `data` do `Node` como 10.

O operador `delete` executa o destrutor `Node` e desaloca a memória alocada com `new` — a memória é retornada ao sistema para que ela possa ser futuramente realocada. Para liberar memória dinamicamente alocada pelo `new` anterior, utilize a instrução

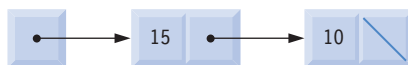
```
delete newPtr;
```

Observe que `newPtr` em si não é excluído; em vez disso, é o espaço para o qual `newPtr` aponta que é excluído. Se ponteiro `newPtr` tem o valor de ponteiro nulo 0, a instrução precedente não tem nenhum efeito. Não é um erro excluir (`delete`) um ponteiro nulo.

As seções a seguir discutem listas, pilhas, filas e árvores. As estruturas de dados apresentadas neste capítulo são criadas e mantidas com alocação dinâmica de memória, classes auto-referenciais, templates de classe e templates de função.

## 21.4 Listas vinculadas

Uma lista vinculada é uma coleção linear de objetos auto-referenciais de classe, chamados **nós**, conectados por **vínculos de ponteiro** — daí o termo lista ‘vinculada’. Uma lista vinculada é acessada via ponteiro ao primeiro nó da lista. Cada nó subsequente é acessado via



**Figura 21.1** Dois objetos de classe auto-referencial vinculados entre si.

o membro ponteiro de vínculo armazenado no nó anterior. Por convenção, o ponteiro de vínculo no último nó de uma lista é configurado como nulo (0) para marcar o final da lista. Os dados são armazenados em uma lista vinculada dinamicamente — cada nó é criado conforme necessário. Um nó pode conter dados de qualquer tipo, incluindo objetos de outras classes. Se os nós contiverem ponteiros de classe básica para objetos de classe básica e classe derivada relacionados por herança, podemos ter uma lista vinculada desses nós e utilizar chamadas de função `virtual` para processar esses objetos polimorficamente. Pilhas e filas também são **estruturas de dados lineares** e, como veremos, podem ser visualizadas como versões restritas de listas vinculadas. As árvores são **estruturas de dados não-lineares**.

As listas de dados podem ser armazenadas em arrays, mas as listas vinculadas fornecem várias vantagens. Uma lista vinculada é apropriada quando o número de elementos de dados a serem representados em qualquer dado momento é imprevisível. As listas vinculadas são dinâmicas, portanto o comprimento de uma lista pode aumentar ou diminuir conforme necessário. O tamanho de um array C++ ‘convencional’, porém, não pode ser alterado, porque o tamanho do array é fixado em tempo de compilação. Os arrays ‘convencionais’ podem tornar-se cheios. As listas vinculadas tornam-se cheias apenas quando o sistema tem memória insuficiente para satisfazer solicitações de alocação de armazenamento dinâmico.



### Dica de desempenho 21.1

*Um array pode ser declarado para conter mais elementos do que o número de itens esperado, mas isso pode desperdiçar memória. As listas vinculadas podem fornecer melhor utilização de memória nessas situações. As listas vinculadas permitem ao programa adaptar em tempo de execução. Observe que o template da classe `vector` (introduzido na Seção 7.11) implementa uma estrutura de dados baseada em array dinamicamente redimensionável.*

As listas vinculadas podem ser mantidas em ordem classificada inserindo cada novo elemento no ponto adequado na lista. Os elementos existentes da lista não precisam ser movidos.



### Dica de desempenho 21.2

*A inserção e a exclusão em um array classificado podem consumir muito tempo — todos os elementos que se seguem ao elemento inserido ou excluído devem ser deslocados apropriadamente. Uma lista vinculada permite operações de inserção eficientes em qualquer lugar da lista.*



### Dica de desempenho 21.3

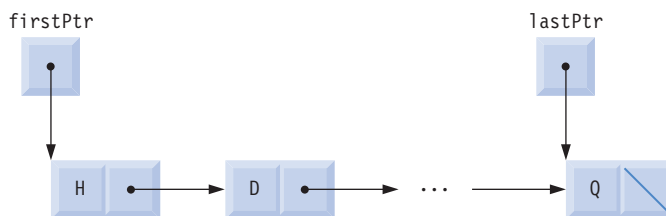
*Os elementos de um array são armazenados contiguamente na memória. Isso permite acesso imediato a qualquer elemento do array, porque o endereço de qualquer elemento pode ser calculado diretamente com base em sua posição em relação ao começo do array. As listas vinculadas não têm recursos para suportar esse ‘acesso direto’ imediato aos seus elementos. Então, acessar elementos individuais em uma lista vinculada pode ser consideravelmente mais caro que acessar elementos individuais em um array. A seleção de uma estrutura de dados é, em geral, baseada no desempenho de operações específicas utilizadas por um programa e na ordem em que os itens de dados são mantidos na estrutura de dados. Por exemplo, normalmente é mais eficiente inserir um item em uma lista vinculada classificada do que em um array classificado.*

Nós de lista vinculada normalmente não são armazenados contiguamente na memória. Logicamente, porém, os nós de uma lista vinculada parecem ser contíguos. A Figura 21.2 ilustra uma lista vinculada com vários nós.



### Dica de desempenho 21.4

*Utilizar alocação dinâmica de memória (em vez de arrays de tamanho fixo) para estruturas de dados que crescem e encolhem em tempo de execução pode poupar memória. Tenha em mente, porém, que os ponteiros ocupam espaço e que a alocação dinâmica de memória incorre no overhead de chamadas de função.*



**Figura 21.2** Representação gráfica de uma lista.

*Implementação de lista vinculada*

O programa das figuras 21.3–21.5 utiliza um template de classe `List` (ver Capítulo 14 para obter informações sobre templates de classe) para manipular uma lista de valores de inteiro e uma lista de valores de ponto flutuante. O programa de driver (Figura 21.5) fornece cinco opções: 1) inserir um valor no começo da lista, 2) inserir um valor no final da lista, 3) excluir um valor do início da lista, 4) excluir um valor do fim da lista e 5) terminar o processamento da lista. Uma discussão detalhada do programa se segue. O Exercício 21.20 pede para você implementar uma função recursiva que imprime uma lista vinculada de trás para a frente, e o Exercício 21.21 pede para você implementar uma função recursiva que pesquisa em uma lista vinculada um item de dados particular.

O programa utiliza os templates de classe `ListNode` (Figura 21.3) e `List` (Figura 21.4). Encapsulada em cada objeto `List` está uma lista vinculada de objetos `ListNode`. O template de classe `ListNode` (Figura 21.3) contém os membros `private` `data` e `nextPtr` (linhas 19–20), um construtor para inicializar esses membros e a função `getData` para retornar os dados em um nó. O membro `data` armazena um valor de tipo `NODETYPE`, o parâmetro de tipo passado ao template de classe. O membro `nextPtr` armazena um ponteiro para o próximo objeto `ListNode` na lista vinculada. Observe que a linha 13 da definição de template de classe `ListNode` declara a classe `List< NODETYPE >` como um `friend`. Isso torna todas as funções-membro de uma dada especialização de template da classe `List` amigas (friends) da especialização correspondente do template de classe `ListNode`, então elas podem acessar os membros `private` de objetos `ListNode` desse tipo. Como o parâmetro `NODETYPE` do template `ListNode` é utilizado como o argumento de template para `List`

```

1 // Figura 21.3: ListNode.h
2 // Definição do template de classe ListNode.
3 #ifndef LISTNODE_H
4 #define LISTNODE_H
5
6 // declaração antecipada da classe List necessária para anunciar essa classe
7 // List existe, portanto pode ser utilizada na declaração friend na linha 13
8 template< typename NODETYPE > class List;
9
10 template< typename NODETYPE>
11 class ListNode
12 {
13 friend class List< NODETYPE >; // torna List uma amiga (friend)
14
15 public:
16 ListNode(const NODETYPE &); // construtor
17 NODETYPE getData() const; // retorna dados no nó
18 private:
19 NODETYPE data; // dados
20 ListNode< NODETYPE > *nextPtr; // próximo nó na lista
21 }; // fim da classe ListNode
22
23 // construtor
24 template< typename NODETYPE>
25 ListNode< NODETYPE >::ListNode(const NODETYPE &info)
26 : data(info), nextPtr(0)
27 {
28 // corpo vazio
29 } // fim do construtor ListNode
30
31 // retorna cópia de dados no nó
32 template< typename NODETYPE >
33 NODETYPE ListNode< NODETYPE >::getData() const
34 {
35 return data;
36 } // fim da função getData
37
38 #endif

```

**Figura 21.3** Definição do template de classe `ListNode`.

na declaração friend, os ListNodes especializados com um tipo particular podem ser processados somente por uma List especializada com o mesmo tipo (por exemplo, uma List de valores int gerencia os objetos ListNode que armazenam os valores int).

As linhas 24–25 do template de classe List (Figura 21.4) declaram os membros de dados private firstPtr (um ponteiro para o primeiro ListNode em uma List) e lastPtr (um ponteiro para o último ListNode em uma List). O construtor-padrão (linhas 32–37) inicializa ambos os ponteiros como 0 (nulo). O destrutor (linhas 40–60) assegura que todos os objetos ListNode em um objeto List são destruídos quando esse objeto List é destruído. As principais funções List são insertAtFront (linhas 63–75), insertAtBack (linhas 78–90), removeFromFront (linhas 93–111) e removeFromBack (linhas 114–141).

A função isEmpty (linhas 144–148) é chamada de função de predicado — não altera a List; em vez disso, determina se a List está vazia (isto é, o ponteiro para o primeiro nó da List é nulo). Se a List estiver vazia, true é retornado; caso contrário, false é retornado. A função print (linhas 159–179) exibe o conteúdo da List. A função utilitária getNode (linhas 151–156) retorna um objeto List-Node alocado dinamicamente. Essa função é chamada a partir das funções insertAtFront e insertAtBack.

```

1 // Figura 21.4: List.h
2 // Definição do template de classe List.
3 #ifndef LIST_H
4 #define LIST_H
5
6 #include <iostream>
7 using std::cout;
8
9 #include "listnode.h" // definição da classe ListNode
10
11 template< typename NODETYPE >
12 class List
13 {
14 public:
15 List(); // construtor
16 ~List(); // destrutor
17 void insertAtFront(const NODETYPE &);
18 void insertAtBack(const NODETYPE &);
19 bool removeFromFront(NODETYPE &);
20 bool removeFromBack(NODETYPE &);
21 bool isEmpty() const;
22 void print() const;
23 private:
24 ListNode< NODETYPE > *firstPtr; // ponteiro para o primeiro nó
25 ListNode< NODETYPE > *lastPtr; // ponteiro para o último nó
26
27 // função utilitária para alocar novo nó
28 ListNode< NODETYPE > *getNode(const NODETYPE &);
29 }; // fim da classe List
30
31 // construtor-padrão
32 template< typename NODETYPE >
33 List< NODETYPE >::List()
34 : firstPtr(0), lastPtr(0)
35 {
36 // corpo vazio
37 } // fim do construtor List
38
39 // destrutor
40 template< typename NODETYPE >
41 List< NODETYPE >::~~List()
42 {
43 if (!isEmpty()) // List não está vazia

```

Figura 21.4 Definição do template de classe List.

(continua)

```

44 {
45 cout << "Destroying nodes ...\n";
46
47 ListNode< NODETYPE > *currentPtr = firstPtr;
48 ListNode< NODETYPE > *tempPtr;
49
50 while (currentPtr != 0) // exclui nós restantes
51 {
52 tempPtr = currentPtr;
53 cout << tempPtr->data << '\n';
54 currentPtr = currentPtr->nextPtr;
55 delete tempPtr;
56 } // fim do while
57 } // fim do if
58
59 cout << "All nodes destroyed\n\n";
60 } // fim do destrutor List
61
62 // insere nó na frente da lista
63 template< typename NODETYPE >
64 void List< NODETYPE >::insertAtFront(const NODETYPE &value)
65 {
66 ListNode< NODETYPE > *newPtr = getNewNode(value); // novo nó
67
68 if (isEmpty()) // List está vazia
69 firstPtr = lastPtr = newPtr; // nova lista tem apenas um nó
70 else // List não está vazia
71 {
72 newPtr->nextPtr = firstPtr; // aponta novo nó para o primeiro nó anterior
73 firstPtr = newPtr; // aponta firstPtr para o novo nó
74 } // fim do else
75 } // fim da função insertAtFront
76
77 // insere nó no fim da lista
78 template< typename NODETYPE >
79 void List< NODETYPE >::insertAtBack(const NODETYPE &value)
80 {
81 ListNode< NODETYPE > *newPtr = getNewNode(value); // novo nó
82
83 if (isEmpty()) // List está vazia
84 firstPtr = lastPtr = newPtr; // nova lista tem apenas um nó
85 else // List não está vazia
86 {
87 lastPtr->nextPtr = newPtr; // atualiza o último nó anterior
88 lastPtr = newPtr; // novo último nó
89 } // fim do else
90 } // fim da função insertAtBack
91
92 // exclui nó da frente da lista
93 template< typename NODETYPE >
94 bool List< NODETYPE >::removeFromFront(NODETYPE &value)
95 {
96 if (isEmpty()) // List está vazia
97 return false; // exclusão malsucedida
98 else
99 {

```

Figura 21.4 Definição do template de classe List.

(continua)

```

100 ListNode< NODETYPE > *tempPtr = firstPtr; // armazena tempPtr para excluir
101
102 if (firstPtr == lastPtr)
103 firstPtr = lastPtr = 0; // nenhum nó permanece depois da exclusão
104 else
105 firstPtr = firstPtr->nextPtr; // aponta para segundo nó anterior
106
107 value = tempPtr->data; // retorna os dados sendo removidos
108 delete tempPtr; // reivindica nó frontal anterior
109 return true; // exclusão bem-sucedida
110 } // fim do else
111 } // fim da função removeFromFront
112
113 // exclui nó do fim da lista
114 template< typename NODETYPE >
115 bool List< NODETYPE >::removeFromBack(NODETYPE &value)
116 {
117 if (isEmpty()) // List está vazia
118 return false; // exclusão malsucedida
119 else
120 {
121 ListNode< NODETYPE > *tempPtr = lastPtr; // armazena tempPtr para excluir
122
123 if (firstPtr == lastPtr) // List tem um elemento
124 firstPtr = lastPtr = 0; // nenhum nó permanece depois da exclusão
125 else
126 {
127 ListNode< NODETYPE > *currentPtr = firstPtr;
128
129 // localiza do segundo ao último elemento
130 while (currentPtr->nextPtr != lastPtr)
131 currentPtr = currentPtr->nextPtr; // move para próximo nó
132
133 lastPtr = currentPtr; // remove último nó
134 currentPtr->nextPtr = 0; // esse é agora o último nó
135 } // fim do else
136
137 value = tempPtr->data; // retorna valor do último nó antigo
138 delete tempPtr; // reivindica o primeiro último nó
139 return true; // exclusão bem-sucedida
140 } // fim do else
141 } // fim da função removeFromBack
142
143 // List está vazia?
144 template< typename NODETYPE >
145 bool List< NODETYPE >::isEmpty() const
146 {
147 return firstPtr == 0;
148 } // fim da função isEmpty
149
150 // retorna ponteiro para nó recentemente alocado
151 template< typename NODETYPE >
152 ListNode< NODETYPE > *List< NODETYPE >::getNewNode(
153 const NODETYPE &value)
154 {
155 return new ListNode< NODETYPE >(value);

```

Figura 21.4 Definição do template de classe List.

(continua)

```

156 } // fim da função getNewNode
157
158 // exibe o conteúdo de List
159 template< typename NODETYPE >
160 void List< NODETYPE >::print() const
161 {
162 if (isEmpty()) // List está vazia
163 {
164 cout << "The list is empty\n\n";
165 return;
166 } // fim do if
167
168 ListNode< NODETYPE > *currentPtr = firstPtr;
169
170 cout << "The list is: ";
171
172 while (currentPtr != 0) // obtém dados de elemento
173 {
174 cout << currentPtr->data << ' ';
175 currentPtr = currentPtr->nextPtr;
176 } // fim do while
177
178 cout << "\n\n";
179 } // fim da função print
180
181 #endif

```

Figura 21.4 Definição do template de classe List.

(continuação)



### Dica de prevenção de erro 21.1

Atribua nulo (0) ao membro de vínculo de um novo nó. Os ponteiros devem ser inicializados antes de ser utilizados.

O programa de driver (Figura 21.5) utiliza o template de função `testList` para permitir ao usuário manipular objetos da classe `List`. As linhas 74 e 78 criam objetos `List` para tipos `int` e `double`, respectivamente. As linhas 75 e 79 invocam o template de função `testList` com esses objetos `List`.

```

1 // Figura 21.5: Fig21_05.cpp
2 // Programa de teste da classe List.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6 using std::endl;
7
8 #include <string>
9 using std::string;
10
11 #include "List.h" // definição da classe List
12
13 // função para testar um List
14 template< typename T >
15 void testList(List< T > &listObject, const string &typeName)
16 {

```

Figura 21.5 Manipulando uma lista vinculada.

(continua)



```

17 cout << "Testing a List of " << typeName << " values\n";
18 instructions(); // exibe instruções
19
20 int choice; // armazena a escolha do usuário
21 T value; // armazena valor de entrada
22
23 do // realiza ações selecionadas pelo usuário
24 {
25 cout << "? ";
26 cin >> choice;
27
28 switch (choice)
29 {
30 case 1: // insere no início
31 cout << "Enter " << typeName << ": ";
32 cin >> value;
33 listObject.insertAtFront(value);
34 listObject.print();
35 break;
36 case 2: // insere no final
37 cout << "Enter " << typeName << ": ";
38 cin >> value;
39 listObject.insertAtBack(value);
40 listObject.print();
41 break;
42 case 3: // remove do início
43 if (listObject.removeFromFront(value))
44 cout << value << " removed from list\n";
45
46 listObject.print();
47 break;
48 case 4: // remove do final
49 if (listObject.removeFromBack(value))
50 cout << value << " removed from list\n";
51
52 listObject.print();
53 break;
54 } // fim do switch
55 } while (choice != 5); // fim da instrução do...while
56
57 cout << "End list test\n\n";
58 } // fim da função testList
59
60 // exibe instruções de programa para o usuário
61 void instructions()
62 {
63 cout << "Enter one of the following:\n"
64 << " 1 to insert at beginning of list\n"
65 << " 2 to insert at end of list\n"
66 << " 3 to delete from beginning of list\n"
67 << " 4 to delete from end of list\n"
68 << " 5 to end list processing\n";
69 } // fim da função instructions
70
71 int main()
72 {

```

Figura 21.5 Manipulando uma lista vinculada.

(continua)

```

73 // testa List de valores int
74 List< int > integerList;
75 testList(integerList, "integer");
76
77 // testa List de valores double
78 List< double > doubleList;
79 testList(doubleList, "double");
80 return 0;
81 } // fim do main

```

Testing a List of integer values

Enter one of the following:

- 1 to insert at beginning of list
- 2 to insert at end of list
- 3 to delete from beginning of list
- 4 to delete from end of list
- 5 to end list processing

? 1

Enter integer: 1

The list is: 1

? 1

Enter integer: 2

The list is: 2 1

? 2

Enter integer: 3

The list is: 2 1 3

? 2

Enter integer: 4

The list is: 2 1 3 4

? 3

2 removed from list

The list is: 1 3 4

? 3

1 removed from list

The list is: 3 4

? 4

4 removed from list

The list is: 3

? 4

3 removed from list

The list is empty

? 5

End list test

Testing a List of double values

Enter one of the following:

- 1 to insert at beginning of list
- 2 to insert at end of list

**Figura 21.5** Manipulando uma lista vinculada.

(continua)

```

3 to delete from beginning of list
4 to delete from end of list
5 to end list processing
? 1
Enter double: 1.1
The list is: 1.1

? 1
Enter double: 2.2
The list is: 2.2 1.1

? 2
Enter double: 3.3
The list is: 2.2 1.1 3.3

? 2
Enter double: 4.4
The list is: 2.2 1.1 3.3 4.4

? 3
2.2 removed from list
The list is: 1.1 3.3 4.4

? 3
1.1 removed from list
The list is: 3.3 4.4

? 4
4.4 removed from list
The list is: 3.3

? 4
3.3 removed from list
The list is empty

? 5
End list test

All nodes destroyed

All nodes destroyed

```

**Figura 21.5** Manipulando uma lista vinculada.

(continuação)

### Função-membro *insertAtFront*

Nas várias páginas a seguir, discutimos cada uma das funções-membro da classe *List* em detalhes. A função *insertAtFront* (Figura 21.4, linhas 63–75) coloca um novo nó na frente da lista. A função consiste em vários passos:

1. Chame a função *getNode* (linha 66), passando para ela o *value*, que é uma referência constante ao valor do nó a ser inserido.
2. A função *getNode* (linhas 151–156) utiliza o operador *new* para criar um novo nó de lista e retorna um ponteiro para esse nó recém-allocado, que é atribuído a *newPtr* em *insertAtFront* (linha 66).
3. Se a lista estiver vazia (linha 68), então tanto *firstPtr* como *lastPtr* são configurados como *newPtr* (linha 69).
4. Se a lista não estiver vazia (linha 70), então o nó apontado por *newPtr* é encadeado na lista copiando *firstPtr* em *newPtr->nextPtr* (linha 72), para que o novo nó aponte ao que costumava ser o primeiro nó da lista e copie *newPtr* em *firstPtr* (linha 73), para que *firstPtr* agora aponte ao novo primeiro nó da lista.

A Figura 21.6 ilustra a função `insertAtFront`. A parte (a) da figura mostra a lista e o novo nó antes da operação `insertAtFront`. As setas tracejadas na parte (b) ilustram o Passo 4 da operação `insertAtFront` que permite ao nó contendo 12 tornar-se a nova frente da lista.

### Função-membro `insertAtBack`

A função `insertAtBack` (Figura 21.4, linhas 78–90) coloca um novo nó no fim da lista. A função consiste em vários passos:

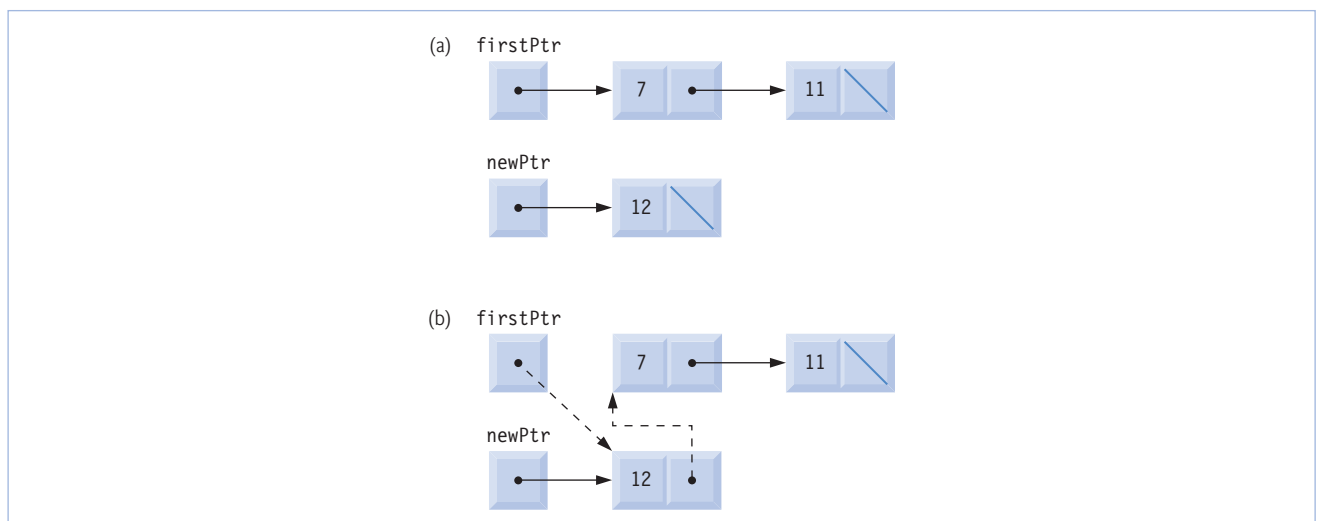
1. Chame a função `getNode` (linha 81), passando para ela o `value`, que é uma referência constante ao valor do nó a ser inserido.
2. A função `getNode` (linhas 151–156) utiliza o operador `new` para criar um novo nó de lista e retorna um ponteiro para esse nó recém-allocado, que é atribuído a `newPtr` em `insertAtBack` (linha 81).
3. Se a lista estiver vazia (linha 83), então tanto `firstPtr` como `lastPtr` são configurados como `newPtr` (linha 84).
4. Se a lista não estiver vazia (linha 85), então o nó apontado por `newPtr` é encadeado na lista copiando `newPtr` em `lastPtr->nextPtr` (linha 87), para que o novo nó seja apontado pelo que costumava ser o último nó da lista e copiando `newPtr` em `lastPtr` (linha 88), para que `lastPtr` agora aponte ao novo último nó da lista.

A Figura 21.7 ilustra uma operação `insertAtBack`. A parte (a) da figura mostra a lista e o novo nó antes da operação. As setas tracejadas na parte (b) ilustram o Passo 4 da função `insertAtBack` que permite a um novo nó ser adicionado ao final de uma lista que não está vazia.

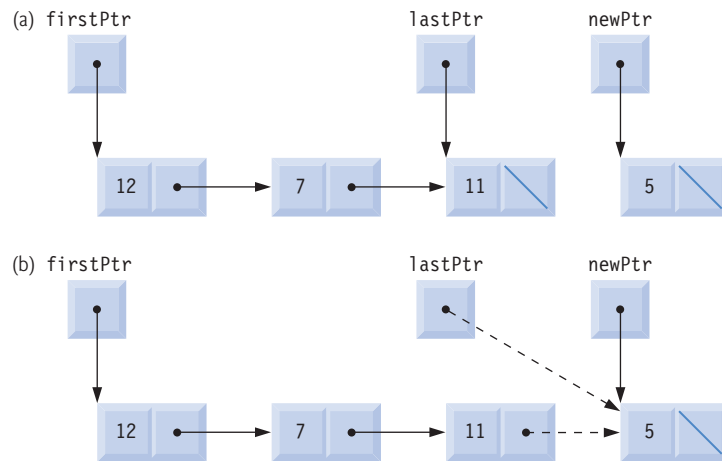
### Função-membro `removeFromFront`

A função `removeFromFront` (Figura 21.4, linhas 93–111) remove o nó frontal da lista e copia o valor de nó no parâmetro de referência. A função retorna `false` se uma tentativa de remover um nó de uma lista vazia for feita (linhas 96–97) e retorna `true` se a remoção for bem-sucedida. A função consiste em vários passos:

1. Atribua a `tempPtr` o endereço para qual `firstPtr` aponta (linha 100). Por fim, `tempPtr` será utilizado para excluir o nó sendo removido.
2. Se `firstPtr` for igual a `lastPtr` (linha 102), isto é, se a lista tiver apenas um elemento antes da tentativa de remoção, então configure `firstPtr` e `lastPtr` como zero (linha 103) para desencadear esse nó da lista (deixando a lista vazia).
3. Se a lista tiver mais de um nó antes da remoção, então deixe `lastPtr` como está e configure `firstPtr` como `firstPtr->nextPtr` (linha 105); isto é, modifique `firstPtr` para apontar para o que era o segundo nó antes da remoção (e agora é o novo primeiro nó).
4. Depois que todas essas manipulações de ponteiro estiverem completas, copie para o parâmetro de referência `value` o membro `data` do nó sendo removido (linha 107).
5. Agora exclua (`delete`) o nó apontado por `tempPtr` (linha 108).
6. Retorne `true`, que indica a remoção bem-sucedida (linha 109).



**Figura 21.6** A operação `insertAtFront` representada graficamente.



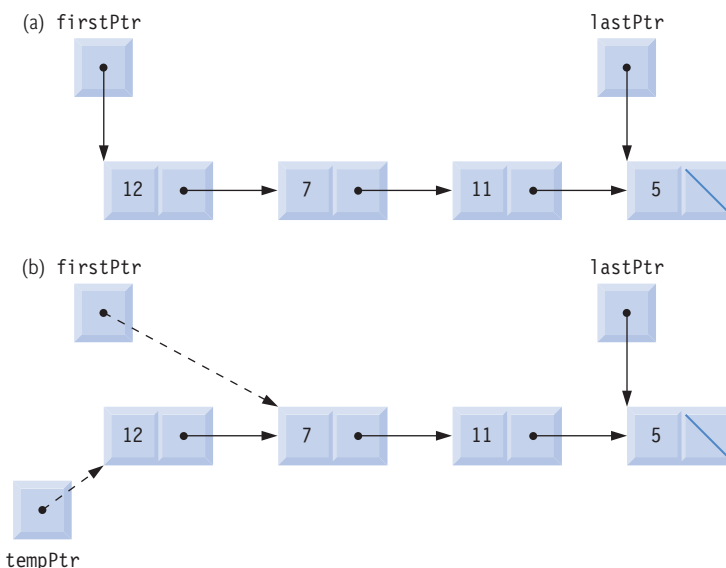
**Figura 21.7** A operação `insertAtBack` representada graficamente.

A Figura 21.8 ilustra a função `removeFromFront`. A parte (a) ilustra a lista antes da operação de remoção. A parte (b) mostra as manipulações reais de ponteiro para remover o nó frontal de uma lista não vazia.

#### Função-membro `removeFromBack`

A função `removeFromBack` (Figura 21.4, linhas 114–141) remove o nó posterior da lista e copia o valor do nó para o parâmetro de referência. A função retorna `false` se uma tentativa de remover um nó de uma lista vazia for feita (linhas 117–118) e retorna `true` se a remoção for bem-sucedida. A função consiste em vários passos:

1. Atribua a `tempPtr` o endereço para o qual `lastPtr` aponta (linha 121). Por fim, `tempPtr` será utilizado para excluir o nó sendo removido.
2. Se `firstPtr` for igual a `lastPtr` (linha 123), isto é, se a lista tiver apenas um elemento antes da tentativa de remoção, então configure `firstPtr` e `lastPtr` como zero (linha 124) para desencadear esse nó da lista (deixando a lista vazia).
3. Se a lista tiver mais de um nó antes da remoção, então atribua a `currentPtr` o endereço para o qual `firstPtr` aponta (linha 127) para ‘percorrer a lista’.



**Figura 21.8** A operação `removeFromFront` representada graficamente.

4. Agora ‘percorra lista’ com `currentPtr` até ela apontar para o nó antes do último nó. Esse nó se tornará o último nó depois que a operação de remoção estiver completa. Isso é feito com um loop `while` (linhas 130–131) que continua substituindo `currentPtr` por `currentPtr->nextPtr`, enquanto `currentPtr->nextPtr` não é `lastPtr`.
5. Atribua `lastPtr` ao endereço para o qual `currentPtr` aponta (linha 133) para desencadear o nó posterior da lista.
6. Configure `currentPtr->nextPtr` como zero (linha 134) no novo último nó da lista.
7. Depois que todas as manipulações de ponteiro forem concluídas, copie para o parâmetro de referência `value` o membro `data` do nó sendo removido (linha 137).
8. Agora exclua (`delete`) o nó apontado por `tempPtr` (linha 138).
9. Retorne `true` (linha 139), que indica que a remoção foi bem-sucedida.

A Figura 21.9 ilustra `removeFromBack`. A parte (a) da figura ilustra a lista antes da operação de remoção. A parte (b) da figura mostra as manipulações de ponteiro reais.

### Função-membro `print`

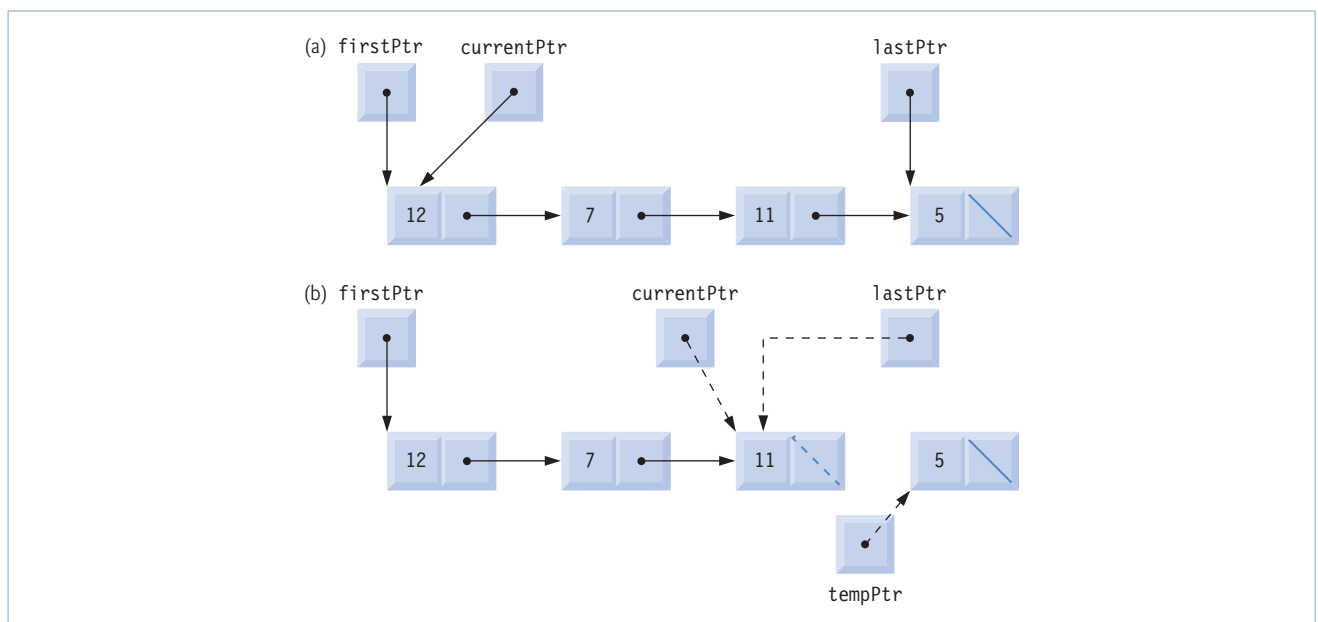
A função `print` (linhas 159–179) primeiro determina se a lista está vazia (linha 162). Se estiver, imprime "The list is empty" e retorna (linhas 164–165). Caso contrário, itera pela lista e gera saída do valor em cada nó. A função inicializa `currentPtr` como uma cópia de `firstPtr` (linha 168), então imprime a string "The list is: " (linha 170). Enquanto `currentPtr` não for nulo (linha 172), `currentPtr->data` é impresso (linha 174) e `currentPtr` recebe o valor de `currentPtr->nextPtr` (linha 175). Observe que, se o vínculo no último nó da lista não for nulo, o algoritmo de impressão imprimirá erroneamente antes do final da lista. O algoritmo de impressão é idêntico para listas vinculadas, pilhas e filas (porque baseamos cada uma dessas estruturas de dados na mesma infra-estrutura de lista vinculada).

### Listas lineares e circulares simples e duplamente vinculadas

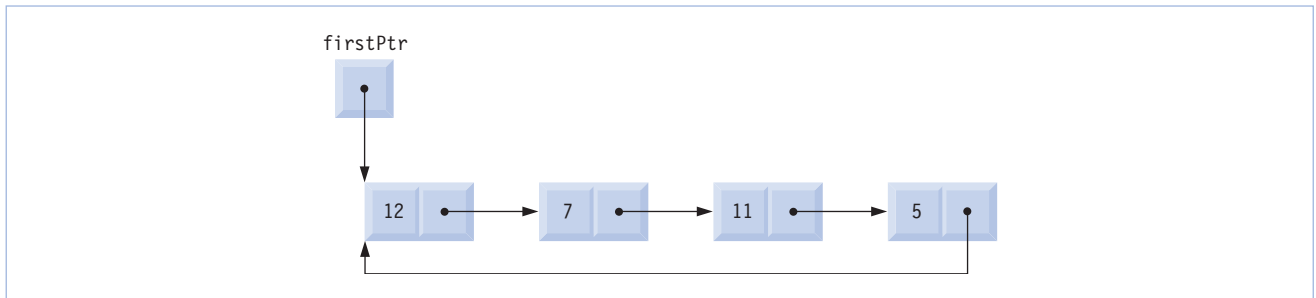
O tipo de lista vinculada que discutimos é uma **lista simplesmente vinculada** — a lista inicia com um ponteiro para o primeiro nó e cada nó contém um ponteiro para o próximo nó ‘na sequência’. Essa lista termina com um nó cujo membro ponteiro tem o valor 0. Uma lista simplesmente vinculada pode ser percorrida em apenas uma direção.

Uma **lista circular simplesmente vinculada** (Figura 21.10) inicia com um ponteiro para o primeiro nó e cada nó contém um ponteiro para o próximo nó. O ‘último nó’ não contém um ponteiro 0; em vez disso, o ponteiro no último nó aponta de volta para o primeiro nó, fechando, assim, o ‘círculo’.

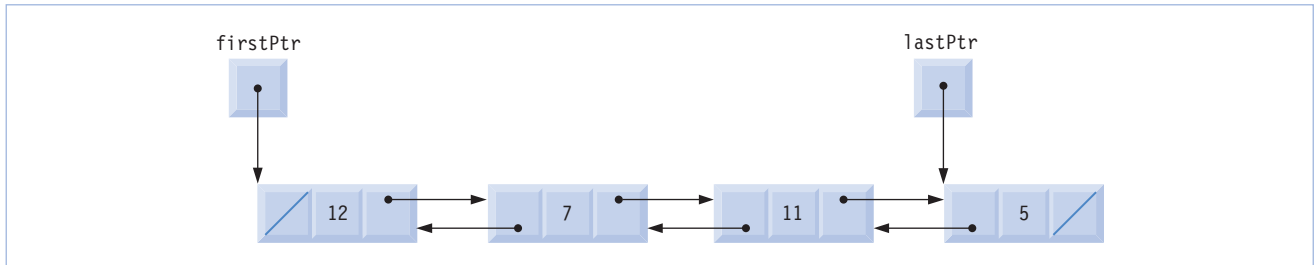
Uma **lista duplamente vinculada** (Figura 21.11) permite percorrer para a frente e para trás. Tal lista é freqüentemente implementada com dois ‘ponteiros iniciais’ — um que aponta para o primeiro elemento da lista para permitir o percurso da frente para trás da lista e um que aponta para o último elemento para permitir o percurso de trás para a frente. Cada nó tem tanto um ponteiro para a frente para o próximo nó na lista na direção para a frente como um ponteiro para trás para o próximo nó na lista na direção para trás. Se sua lista



**Figura 21.9** A operação `removeFromBack` representada graficamente.



**Figura 21.10** Lista circular simplesmente vinculada.



**Figura 21.11** Lista duplamente vinculada.

contém um diretório de telefones indexados em ordem alfabética, por exemplo, uma pesquisa por alguém cujo nome inicia com uma letra próxima do início do alfabeto poderia começar a partir do topo da lista. Procurar alguém cujo nome inicia com uma letra próxima do fim do alfabeto poderia começar a partir do final da lista.

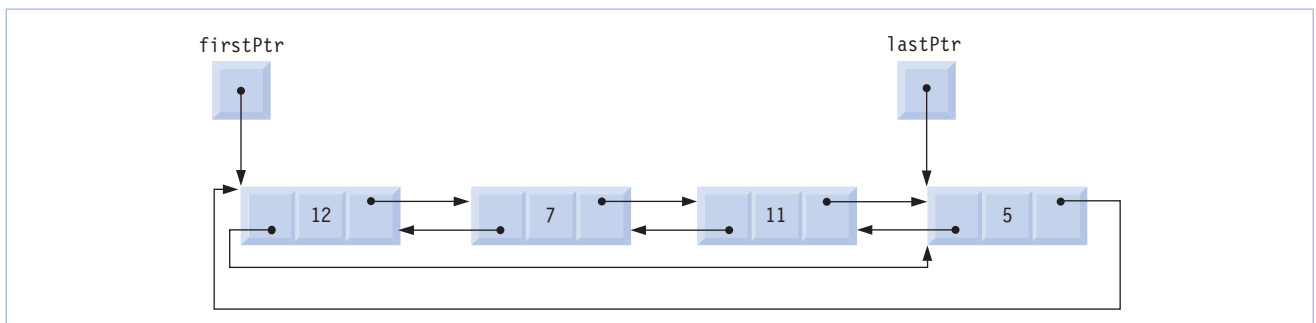
Em uma **lista circular duplamente vinculada** (Figura 21.12), o ponteiro para a frente da lista do último nó aponta para o primeiro nó e o ponteiro para trás do primeiro nó aponta para o último nó, fechando assim o ‘círculo’.

## 21.5 Pilhas

No Capítulo 14, “Templates”, explicamos a noção de um template de classe de pilhas com uma implementação de array subjacente. Nesta seção, utilizamos uma implementação de lista vinculada baseada em ponteiro subjacente. Discutimos também as pilhas no Capítulo 23, “Standard Template Library (STL)”.

Uma estrutura de dados de pilha permite que nós sejam adicionados à pilha e removidos da pilha somente na parte superior. Por essa razão, uma pilha é referida como uma estrutura de dados primeiro a entrar, primeiro a sair (*last-in, first-out* – LIFO). Uma maneira de implementar uma pilha é como uma versão restrita de uma lista vinculada. Em tal implementação, o membro de vínculo no último nó da pilha é configurado como nulo (zero) para indicar a parte inferior da pilha.

As principais funções-membro utilizadas para manipular uma pilha são `push` e `pop`. A função `push` insere um novo nó na parte superior da pilha. A função `pop` remove um nó da parte superior da pilha, armazena o valor removido em uma variável de referência que é passada para função chamadora e retorna `true` se a operação `pop` foi bem-sucedida (`false`, caso contrário).



**Figura 21.12** Lista circular duplamente vinculada.



As pilhas têm muitas aplicações interessantes. Por exemplo, quando uma chamada de função é feita, a função chamada deve saber retornar para seu chamador, portanto o endereço de retorno é inserido em uma pilha. Se uma série de chamadas de função ocorre, os valores sucessivos de retorno são inseridos na pilha na ordem primeiro a entrar, primeiro a sair, de modo que cada função possa retornar para seu chamador. As pilhas suportam chamadas de função recursiva da mesma maneira que as chamadas não recursivas convencionais. A Seção 6.11 discute a pilha de chamadas de função em detalhes.

As pilhas fornecem a memória para, e armazenam os valores de, variáveis automáticas em cada invocação de uma função. Quando a função retorna para seu chamador ou lança uma exceção, o destrutor (se houver algum) de cada objeto local é chamado, o espaço para as variáveis automáticas dessa função é removido da pilha e essas variáveis não são mais conhecidas pelo programa.

As pilhas são utilizadas por compiladores no processo de avaliar expressões e gerar código de linguagem de máquina. Os exercícios exploram várias aplicações de pilhas, inclusive utilizá-las para desenvolver seu próprio compilador funcional completo.

Tiramos proveito do íntimo relacionamento entre listas e pilhas para implementar uma classe de pilha principalmente reutilizando uma classe de lista. Primeiro, implementamos a classe de pilha por herança *private* da classe `List`. Então implementamos uma classe de pilha de idêntico desempenho por meio de composição incluindo um objeto lista como um membro *private* de uma classe de pilhas. Naturalmente, todas as estruturas de dados deste capítulo, incluindo essas duas classes de pilha, são implementadas como templates para encorajar mais reusabilidade.

O programa das figuras 21.13–21.14 cria um template de classe `Stack` (Figura 21.13) principalmente por meio da herança *private* (linha 9) do template de classe `List` da Figura 21.4. Queremos que a `Stack` tenha funções-membro `push` (linhas 13–16), `pop` (linhas

```

1 // Figura 21.13: Stack.h
2 // Definição do template de classe Stack derivada da classe List.
3 #ifndef STACK_H
4 #define STACK_H
5
6 #include "List.h" // definição da classe List
7
8 template< typename STACKTYPE >
9 class Stack : private List< STACKTYPE >
10 {
11 public:
12 // push chama a função List insertAtFront
13 void push(const STACKTYPE &data)
14 {
15 insertAtFront(data);
16 } // fim da função push
17
18 // pop chama a função List removeFromFront
19 bool pop(STACKTYPE &data)
20 {
21 return removeFromFront(data);
22 } // fim da função pop
23
24 // isEmpty chama a função List isEmpty
25 bool isEmpty() const
26 {
27 return isEmpty();
28 } // fim da função isEmpty
29
30 // printStack chama a função List print
31 void printStack() const
32 {
33 print();
34 } // fim da função print
35 }; // fim da classe Stack
36
37 #endif

```

**Figura 21.13** Definição do template de classe `Stack`.

19–22), `isEmpty` (linhas 25–28) e `printStack` (linhas 31–34). Observe que essas são essencialmente as funções `insertAtFront`, `removeFromFront`, `isEmpty` e `print` do template da classe `List`. Naturalmente, o template de classe `List` contém outras funções-membro (isto é, `insertAtBack` e `removeFromBack`) que não gostaríamos de tornar acessível pela interface `public` para a classe `Stack`. Então, quando indicamos que o template de classe `Stack` deve herdar do template de classe `List`, especificamos a herança `private`. Isso torna `private` todas as funções-membro do template de classe `List` no template de classe `Stack`. Ao implementarmos as funções-membro de `Stack`, fazemos então cada uma dessas funções chamar a função-membro apropriada da classe `List` — `push` chama `insertAtFront` (linha 15), `pop` chama `removeFromFront` (linha 21), `isEmpty` chama `isEmpty` (linha 27) e `printStack` chama `print` (linha 33) — isso é referido como **delegação**.

O template de classe de pilhas é utilizado em `main` (Figura 21.14) para instanciar a pilha de inteiros `intStack` do tipo `Stack< int >` (linha 11). Os inteiros de 0 a 2 são inseridos em `intStack` (linhas 16–20) e, então, removidos de `intStack` (linhas 25–30). O programa utiliza o template da classe `Stack` para criar `doubleStack` do tipo `Stack< double >` (linha 32). Os valores 1.1, 2.2 e 3.3 são inseridos em `doubleStack` (linhas 38–43) e, então, removidos de `doubleStack` (linhas 48–53).

Outra maneira de implementar um template de classe `Stack` é reutilizando o template de classe `List` por meio da composição. A Figura 21.15 é uma nova implementação do template de classe `Stack` que contém um objeto `List< STACKTYPE >` chamado `stackList` (linha 38). Essa versão do template de classe `Stack` utiliza a classe `List` da Figura 21.4. Para testar essa classe, use o programa de driver da Figura 21.14, mas inclua o novo arquivo de cabeçalho — `Stackcomposition.h` na linha 6 desse arquivo. A saída do programa é idêntica para ambas as versões da classe `Stack`.

```

1 // Figura 21.14: Fig21_14.cpp
2 // Programa de teste do template de classe Stack.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Stack.h" // definição da classe Stack
8
9 int main()
10 {
11 Stack< int > intStack; // cria Stack de ints
12
13 cout << "processing an integer Stack" << endl;
14
15 // insere inteiros em intStack
16 for (int i = 0; i < 3; i++)
17 {
18 intStack.push(i);
19 intStack.printStack();
20 } // fim do for
21
22 int popInteger; // armazena o int removido da pilha
23
24 // remove os inteiros de intStack
25 while (!intStack.isEmpty())
26 {
27 intStack.pop(popInteger);
28 cout << popInteger << " popped from stack" << endl;
29 intStack.printStack();
30 } // fim do while
31
32 Stack< double > doubleStack; // cria Stack de doubles
33 double value = 1.1;
34
35 cout << "processing a double Stack" << endl;
36
37 // insere valores de ponto flutuante em doubleStack

```

**Figura 21.14** Um programa simples de pilha.

(continua)

```

38 for (int j = 0; j < 3; j++)
39 {
40 doubleStack.push(value);
41 doubleStack.printStack();
42 value += 1.1;
43 } // fim do for
44
45 double popDouble; // armazena o double removido da pilha
46
47 // remove os valores de ponto flutuante de doubleStack
48 while (!doubleStack.isEmpty())
49 {
50 doubleStack.pop(popDouble);
51 cout << popDouble << " popped from stack" << endl;
52 doubleStack.printStack();
53 } // fim do while
54
55 return 0;
56 } // fim do main

```

processing an integer Stack

The list is: 0

The list is: 1 0

The list is: 2 1 0

2 popped from stack

The list is: 1 0

1 popped from stack

The list is: 0

0 popped from stack

The list is empty

processing a double Stack

The list is: 1.1

The list is: 2.2 1.1

The list is: 3.3 2.2 1.1

3.3 popped from stack

The list is: 2.2 1.1

2.2 popped from stack

The list is: 1.1

1.1 popped from stack

The list is empty

All nodes destroyed

All nodes destroyed

Figura 21.14 Um programa simples de pilha.

(continuação)

```

1 // Figura 21.15: Stackcomposition.h
2 // Definição do template de classe Stack com objeto List composto.
3 #ifndef STACKCOMPOSITION_H
4 #define STACKCOMPOSITION_H
5
6 #include "List.h" // definição da classe List
7
8 template< typename STACKTYPE >
9 class Stack
10 {
11 public:
12 // nenhum construtor; construtor List faz inicialização
13
14 // push chama a função-membro insertAtFront do objeto stackList
15 void push(const STACKTYPE &data)
16 {
17 stackList.insertAtFront(data);
18 } // fim da função push
19
20 // pop chama a função-membro removeFromFront do objeto stackList
21 bool pop(STACKTYPE &data)
22 {
23 return stackList.removeFromFront(data);
24 } // fim da função pop
25
26 // isEmpty chama a função-membro isEmpty do objeto stackList
27 bool isEmpty() const
28 {
29 return stackList.isEmpty();
30 } // fim da função isEmpty
31
32 // printStack chama a função-membro print do objeto stackList
33 void printStack() const
34 {
35 stackList.print();
36 } // fim da função printStack
37 private:
38 List< STACKTYPE > stackList; // objeto List composto
39 }; // fim da classe Stack
40
41 #endif

```

**Figura 21.15** O template de classe Stack com um objeto List composto.

## 21.6 Filas

Uma **fila** (*queue*) é semelhante a uma fila de pessoas no caixa de um supermercado — a primeira pessoa na fila é atendida em primeiro lugar, e outros clientes entram no fim da fila e esperam pelo atendimento. Os nós da fila são removidos apenas do início (ou cabeça) da fila e são inseridos somente no final (ou cauda) da fila. Por essa razão, uma fila é referida como uma estrutura de dados primeiro a entrar, primeiro a sair (*first-in, first-out* – FIFO). As operações de inserção e remoção são conhecidas como **enqueue** e **dequeue**.

As filas têm muitas aplicações em sistemas de computador. Os computadores que têm um único processador podem atender apenas um usuário por vez. As entradas para os outros usuários são colocadas em uma fila. Cada entrada avança gradualmente para a frente da fila quando os usuários recebem o serviço. A entrada na frente da fila é a próxima a receber o serviço.

As filas também são utilizadas para suportar **spooling de impressão**. Por exemplo, uma única impressora talvez seja compartilhada por todos os usuários de uma rede. Muitos usuários podem enviar trabalhos de impressão à impressora, mesmo quando a impressora já estiver ocupada. Esses trabalhos de impressão são colocados em uma fila até a impressora ficar disponível. Um programa chamado **spooler** gerencia a fila para assegurar que, à medida que cada trabalho de impressão é concluído, o próximo trabalho de impressão é enviado à impressora.

Os pacotes de informação também esperam em filas em redes de computadores. Toda vez que um pacote chega a um nó de rede, ele deve ser roteado para o próximo nó na rede ao longo do caminho até o destino final do pacote. O nó de roteamento roteia um pacote por vez, então pacotes adicionais são enfileirados até o roteador conseguir rotá-los.

Um servidor de arquivos em uma rede de computadores trata as solicitações de acesso de arquivo de muitos clientes por toda a rede. Os servidores têm uma capacidade limitada para servir solicitações de clientes. Quando essa capacidade é excedida, as solicitações dos clientes esperam em filas.

O programa das figuras 21.16–21.17 cria um template de classe Queue (Figura 21.16) pela herança private (linha 9) do template de classe List da Figura 21.4. Queremos que a Queue tenha funções-membro enqueue (linhas 13–16), dequeue (linhas 19–22), isEmpty (linhas 25–28) e printQueue (linhas 31–34). Observe que essas são essencialmente as funções insertAtBack, removeFromFront, isEmpty e print do template da classe List. Naturalmente, o template de classe List contém outras funções-membro (isto é, insertAtFront e removeFromBack) que não gostaríamos de tornar acessíveis pela interface public para a classe Queue. Então, quando indicamos que o template de classe Queue deve herdar o template de classe List, especificamos a herança private. Isso torna private todas as funções-membro do template de classe List no template de classe Queue. Quando implementamos as funções-membro de Queue, fazemos cada uma dessas funções chamar a função-membro apropriada da classe de listas — enqueue chama insertAtBack (linha 15), dequeue chama removeFromFront (linha 21), isEmpty chama isEmpty (linha 27) e printQueue chama print (linha 33). Novamente, isso é chamado delegação.

```

1 // Figura 21.16: Queue.h
2 // Definição do template de classe Queue derivada da classe List.
3 #ifndef QUEUE_H
4 #define QUEUE_H
5
6 #include "List.h" // definição da classe List
7
8 template< typename QUEUETYPE >
9 class Queue : private List< QUEUETYPE >
10 {
11 public:
12 // enqueue chama função-membro List insertAtBack
13 void enqueue(const QUEUETYPE &data)
14 {
15 insertAtBack(data);
16 } // fim da função enqueue
17
18 // dequeue chama a função-membro List removeFromFront
19 bool dequeue(QUEUETYPE &data)
20 {
21 return removeFromFront(data);
22 } // fim da função dequeue
23
24 // isEmpty chama a função-membro List isEmpty
25 bool isEmpty() const
26 {
27 return isEmpty();
28 } // fim da função isEmpty
29
30 // printQueue chama função-membro List print
31 void printQueue() const
32 {
33 print();
34 } // fim da função printQueue
35 }; // fim da classe Queue
36
37 #endif

```

**Figura 21.16** Definição do template de classe Queue.

A Figura 21.17 utiliza o template de classe Queue para instanciar a fila de inteiros intQueue do tipo Queue< int > (linha 11). Os inteiros de 0 a 2 são enfileirados para intQueue (linhas 16–20) e, então, desenfileirados de intQueue na ordem primeiro a entrar, primeiro a sair (linhas 25–30). Em seguida, o programa instancia a fila doubleQueue do tipo Queue< double > (linha 32). Os valores 1.1, 2.2 e 3.3 são enfileirados para doubleQueue (linhas 38–43) e, então, desenfileirados de doubleQueue na ordem primeiro a entrar, primeiro a sair (linhas 48–53).

```

1 // Figura 21.17: Fig21_17.cpp
2 // Programa de teste do template de classe Queue.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Queue.h" // definição da classe Queue
8
9 int main()
10 {
11 Queue< int > intQueue; // cria Queue de inteiros
12
13 cout << "processing an integer Queue" << endl;
14
15 // enfileira inteiros em intQueue
16 for (int i = 0; i < 3; i++)
17 {
18 intQueue.enqueue(i);
19 intQueue.printQueue();
20 } // fim do for
21
22 int dequeueInteger; // armazena inteiro desenfileirado
23
24 // desenfileira inteiros de intQueue
25 while (!intQueue.isEmpty())
26 {
27 intQueue.dequeue(dequeueInteger);
28 cout << dequeueInteger << " dequeued" << endl;
29 intQueue.printQueue();
30 } // fim do while
31
32 Queue< double > doubleQueue; // criar Queue de doubles
33 double value = 1.1;
34
35 cout << "processing a double Queue" << endl;
36
37 // enfileira valores de ponto flutuante em doubleQueue
38 for (int j = 0; j < 3; j++)
39 {
40 doubleQueue.enqueue(value);
41 doubleQueue.printQueue();
42 value += 1.1;
43 } // fim do for
44
45 double dequeueDouble; // armazena double desenfileirado
46
47 // desenfileira valores de ponto flutuante de doubleQueue
48 while (!doubleQueue.isEmpty())
49 {

```

Figura 21.17 Programa de processamento de fila.

(continua)

```

50 doubleQueue.dequeue(dequeueDouble);
51 cout << dequeueDouble << " dequeued" << endl;
52 doubleQueue.printQueue();
53 } // fim do while
54
55 return 0;
56 } // fim do main

```

```

processing an integer Queue
The list is: 0

```

```

The list is: 0 1

```

```

The list is: 0 1 2

```

```

0 dequeued
The list is: 1 2

```

```

1 dequeued
The list is: 2

```

```

2 dequeued
The list is empty

```

```

processing a double Queue
The list is: 1.1

```

```

The list is: 1.1 2.2

```

```

The list is: 1.1 2.2 3.3

```

```

1.1 dequeued
The list is: 2.2 3.3

```

```

2.2 dequeued
The list is: 3.3

```

```

3.3 dequeued
The list is empty

```

```

All nodes destroyed

```

```

All nodes destroyed

```

**Figura 21.17** Programa de processamento de fila.

(continuação)

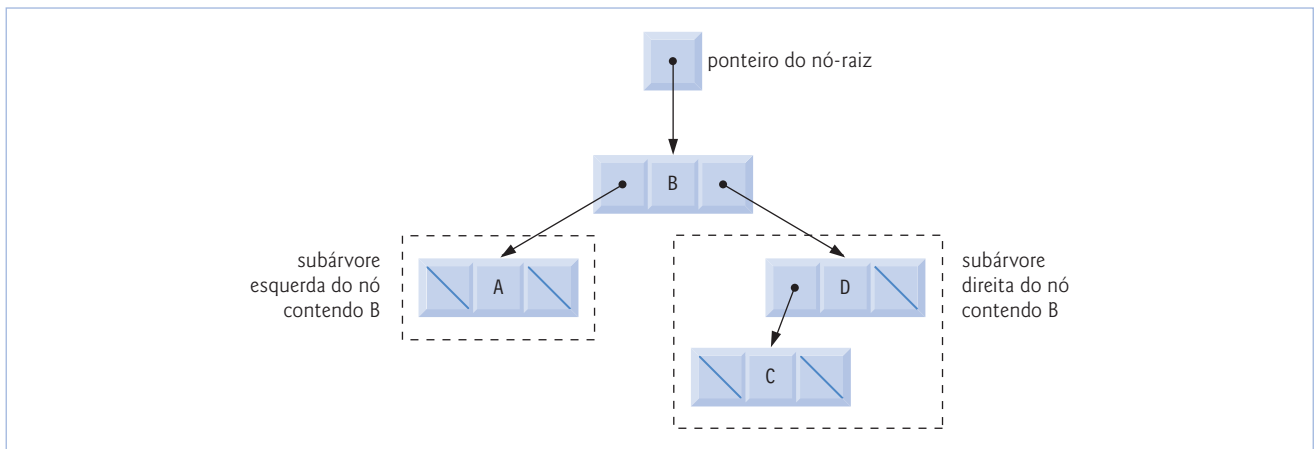
## 21.7 Árvores

As listas vinculadas, pilhas e filas são estruturas de dados lineares. Uma árvore é uma estrutura de dados bidimensional não-linear. Os nós da árvore contêm dois ou mais vínculos. Esta seção discute as **árvores binárias** (Figura 21.18) — árvores cujos nós contêm, cada um, dois vínculos (nenhum, um ou ambos os quais podem ser nulos).

### Terminologia básica

Para os propósitos desta discussão, consulte os nós A, B, C e D na Figura 21.18. O **nó-raiz** (nó b) é o primeiro nó em uma árvore. Cada vínculo no nó-raiz referencia um **filho** (nós A e D). O **filho esquerdo** (nó a) é o nó-raiz da **subárvore esquerda** (que contém apenas o nó a), e o **filho direito** (nó d) é o nó-raiz da **subárvore direita** (que contém os nós D e C). Os filhos de um único nó são chamados **irmãos** (por exemplo, nós A e D são irmãos). Um nó sem filhos é chamado **nó-folha** (por exemplo, nós A e C são nós-folha). Em geral,





**Figura 21.18** Uma representação gráfica de uma árvore binária.

os cientistas da computação normalmente desenhavam árvores a partir do nó-raiz para baixo — exatamente o oposto de como as árvores crescem na natureza.

### Árvores de pesquisa binária

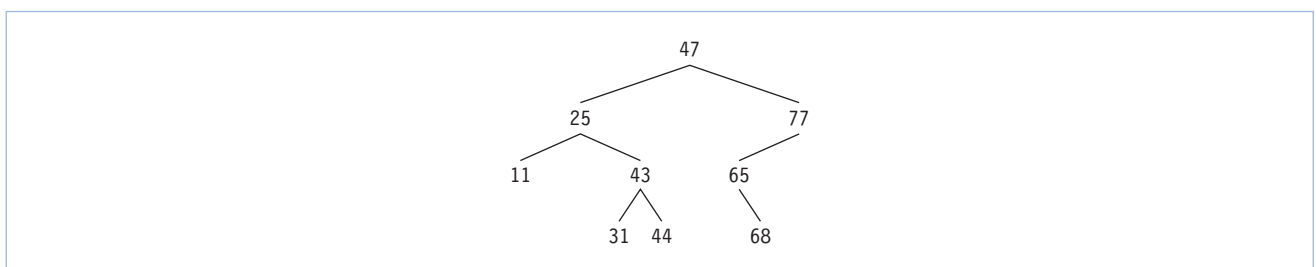
Esta seção discute uma árvore binária especial chamada de **árvore de pesquisa binária**. Uma árvore de pesquisa binária (sem valores de nó duplicados) tem a característica de que os valores em qualquer subárvore esquerda são menores que o valor em seu **nó-pai**, e os valores em qualquer subárvore direita são maiores que o valor em seu nó-pai. A Figura 21.19 ilustra uma árvore de pesquisa binária com 9 valores. Observe que a forma da árvore de pesquisa binária que corresponde a um conjunto de dados pode variar, dependendo da ordem em que os valores são inseridos na árvore.

### Implementando o programa de árvore de pesquisa binária

O programa das figuras 21.20–21.22 cria uma árvore de pesquisa binária e a percorre (isto é, passa por todos os seus nós) de três maneiras — utilizando os percursos recursivos **na ordem**, **pré-ordem** e **pós-ordem**. Explicamos esses algoritmos de percurso em breve.

Começamos nossa discussão com o programa de driver (Figura 21.22), então continuamos com as implementações de classes `TreeNode` (Figura 21.20) e `Tree` (Figura 21.21). A função `main` (Figura 21.22) começa instanciando a árvore de inteiro `intTree` do tipo `Tree< int >` (linha 15). O programa solicita 10 inteiros, cada um dos quais é inserido na árvore binária chamando `insertNode` (linha 24). O programa então realiza os percursos pré-ordem, na ordem e pós-ordem (esses serão explicados em breve) da `intTree` (linhas 28, 31 e 34, respectivamente). O programa então instancia a árvore de ponto flutuante `doubleTree` do tipo `Tree< double >` (linha 36). O programa solicita 10 valores `double`, cada um dos quais é inserido na árvore binária chamando `insertNode` (linha 46). Em seguida, o programa realiza os percursos pré-ordem, na ordem e pós-ordem de `doubleTree` (linhas 50, 53 e 56, respectivamente).

Agora discutiremos as definições do template de classe. Iniciamos com a definição de template de classe `TreeNode` (Figura 21.20) que declara `Tree< NODETYPE >` como seu `friend` (linha 13). Isso torna todas as funções-membro de uma dada especialização de template da classe `Tree` (Figura 21.21) amigas da especialização correspondente do template de classe `TreeNode`. Desse modo elas podem acessar os membros `private` dos objetos `TreeNode` desse tipo. Como o parâmetro `NODETYPE` do template `TreeNode` é utilizado como o argumento de template para `Tree` na declaração `friend`, os `TreeNodes` especializados com um tipo particular podem ser processados somente por uma `Tree` especializada com o mesmo tipo (por exemplo, uma `Tree` de valores `int` gerencia os objetos `TreeNode` que armazenam os valores `int`).



**Figura 21.19** Uma árvore de pesquisa binária.

```

1 // Figura 21.20: Treenode.h
2 // Definição do template de classe TreeNode.
3 #ifndef TREENODE_H
4 #define TREENODE_H
5
6 // encaminha a declaração da classe Tree
7 template< typename NODETYPE > class Tree;
8
9 // definição do template de classe TreeNode
10 template< typename NODETYPE >
11 class TreeNode
12 {
13 friend class Tree< NODETYPE >;
14 public:
15 // construtor
16 TreeNode(const NODETYPE &d)
17 : leftPtr(0), // ponteiro para subárvore esquerda
18 data(d), // dados de nó de árvore
19 rightPtr(0) // ponteiro para subárvore direita
20 {
21 // corpo vazio
22 } // fim do construtor TreeNode
23
24 // retorna a cópia de dados do nó
25 NODETYPE getData() const
26 {
27 return data;
28 } // fim da função getData
29 private:
30 TreeNode< NODETYPE > *leftPtr; // ponteiro para subárvore esquerda
31 NODETYPE data;
32 TreeNode< NODETYPE > *rightPtr; // ponteiro para subárvore direita
33 }; // fim da classe TreeNode
34
35 #endif

```

**Figura 21.20** Definição do template de classe TreeNode.

```

1 // Figura 21.21: Tree.h
2 // Definição do template de classe Tree.
3 #ifndef TREE_H
4 #define TREE_H
5
6 #include <iostream>
7 using std::cout;
8 using std::endl;
9
10 #include "Treenode.h"
11
12 // definição do template de classe Tree
13 template< typename NODETYPE > class Tree
14 {
15 public:
16 Tree(); // construtor

```

**Figura 21.21** Definição do template de classe Tree.

(continua)

```

17 void insertNode(const NODETYPE &);
18 void preOrderTraversal() const;
19 void inOrderTraversal() const;
20 void postOrderTraversal() const;
21 private:
22 TreeNode< NODETYPE > *rootPtr;
23
24 // funções utilitárias
25 void insertNodeHelper(TreeNode< NODETYPE > **, const NODETYPE &);
26 void preOrderHelper(TreeNode< NODETYPE > *) const;
27 void inOrderHelper(TreeNode< NODETYPE > *) const;
28 void postOrderHelper(TreeNode< NODETYPE > *) const;
29 }; // fim da classe Tree
30
31 // construtor
32 template< typename NODETYPE >
33 Tree< NODETYPE >::Tree()
34 {
35 rootPtr = 0; // indica que a árvore está inicialmente vazia
36 } // fim do construtor Tree
37
38 // insere nó em Tree
39 template< typename NODETYPE >
40 void Tree< NODETYPE >::insertNode(const NODETYPE &value)
41 {
42 insertNodeHelper(&rootPtr, value);
43 } // fim da função insertNode
44
45 // função utilitária chamada por insertNode; recebe um ponteiro
46 // para que a função possa modificar o valor do ponteiro
47 template< typename NODETYPE >
48 void Tree< NODETYPE >::insertNodeHelper(
49 TreeNode< NODETYPE > **ptr, const NODETYPE &value)
50 {
51 // subárvore está vazia; cria novo TreeNode contendo o valor
52 if (*ptr == 0)
53 *ptr = new TreeNode< NODETYPE >(value);
54 else // subárvore não está vazia
55 {
56 // os dados a inserir são menores que os dados no nó atual
57 if (value < (*ptr)->data)
58 insertNodeHelper(&(*ptr)->leftPtr, value);
59 else
60 {
61 // os dados a inserir são maiores que os dados no nó atual
62 if (value > (*ptr)->data)
63 insertNodeHelper(&(*ptr)->rightPtr, value);
64 else // duplica valor dos dados ignorados
65 cout << value << " dup" << endl;
66 } // fim do else
67 } // fim do else
68 } // fim da função insertNodeHelper
69
70 // inicia o percurso na pré-ordem de Tree
71 template< typename NODETYPE >
72 void Tree< NODETYPE >::preOrderTraversal() const

```

Figura 21.21 Definição do template de classe Tree.

(continua)

```

73 {
74 preOrderHelper(rootPtr);
75 } // fim da função preOrderTraversal
76
77 // função utilitária para executar o percurso na pré-ordem de Tree
78 template< typename NODETYPE >
79 void Tree< NODETYPE >::preOrderHelper(TreeNode< NODETYPE > *ptr) const
80 {
81 if (ptr != 0)
82 {
83 cout << ptr->data << ' '; // processa nó
84 preOrderHelper(ptr->leftPtr); // percorre subárvore esquerda
85 preOrderHelper(ptr->rightPtr); // percorre subárvore direita
86 } // fim do if
87 } // fim da função preOrderHelper
88
89 // inicia percurso na ordem de Tree
90 template< typename NODETYPE >
91 void Tree< NODETYPE >::inOrderTraversal() const
92 {
93 inOrderHelper(rootPtr);
94 } // fim da função inOrderTraversal
95
96 // função utilitária para realizar o percurso na ordem de Tree
97 template< typename NODETYPE >
98 void Tree< NODETYPE >::inOrderHelper(TreeNode< NODETYPE > *ptr) const
99 {
100 if (ptr != 0)
101 {
102 inOrderHelper(ptr->leftPtr); // percorre subárvore esquerda
103 cout << ptr->data << ' '; // processa nó
104 inOrderHelper(ptr->rightPtr); // percorre subárvore direita
105 } // fim do if
106 } // fim da função inOrderHelper
107
108 // inicia o percurso na pós-ordem de Tree
109 template< typename NODETYPE >
110 void Tree< NODETYPE >::postOrderTraversal() const
111 {
112 postOrderHelper(rootPtr);
113 } // fim da função postOrderTraversal
114
115 // função utilitária para realizar o percurso de pós-ordem de Tree
116 template< typename NODETYPE >
117 void Tree< NODETYPE >::postOrderHelper(
118 TreeNode< NODETYPE > *ptr) const
119 {
120 if (ptr != 0)
121 {
122 postOrderHelper(ptr->leftPtr); // percorre subárvore esquerda
123 postOrderHelper(ptr->rightPtr); // percorre subárvore direita
124 cout << ptr->data << ' '; // processa nó
125 } // fim do if
126 } // fim da função postOrderHelper
127
128 #endif

```

Figura 21.21 Definição do template de classe Tree.

(continuação)

As linhas 30–32 declaram os dados `private` de um `TreeNode` — o valor `data` do nó, e os ponteiros `leftPtr` (para a subárvore esquerda do nó) e `rightPtr` (para a subárvore direita do nó). O construtor (linhas 16–22) configura `data` como o valor fornecido como um argumento de construtor e configura os ponteiros `leftPtr` e `rightPtr` como zero (inicializando, assim, esse nó como um nó-folha). A função-membro `getData` (linhas 25–28) retorna o valor `data`.

O template de classe `Tree` (Figura 21.21) tem como dados `private` `rootPtr` (linha 22), um ponteiro para o nó-raiz da árvore. As linhas 17–20 do template de classe declaram as funções-membro `public` `insertNode` (que insere um novo nó na árvore) e `preOrderTraversal`, `inOrderTraversal` e `postOrderTraversal`, cada uma das quais percorre a árvore da maneira designada. Cada uma dessas funções-membro chama sua própria função utilitária recursiva separada para realizar as operações apropriadas na representação interna da árvore, assim o programa não precisa acessar os dados `private` subjacentes para realizar essas funções. Lembre-se de que a recursão requer que passemos um ponteiro que representa a próxima subárvore a ser processada. O construtor `Tree` inicializa `rootPtr` como zero para indicar que a árvore está inicialmente vazia.

A função utilitária `insertNodeHelper` (linhas 47–68) da classe `Tree` é chamada por `insertNode` (linhas 39–43) para inserir recursivamente um nó na árvore. *Um nó pode ser inserido somente como um nó-folha em uma árvore de pesquisa binária.* Se a árvore está vazia, um novo `TreeNode` é criado, inicializado e inserido na árvore (linhas 52–53).

Se a árvore não está vazia, o programa compara o valor a ser inserido com o valor `data` no nó-raiz. Se o valor de inserção é menor (linha 57), o programa chama `insertNodeHelper` (linha 58) recursivamente para inserir o valor na subárvore esquerda. Se o valor de inserção é maior (linha 62), o programa chama `insertNodeHelper` (linha 63) recursivamente para inserir o valor na subárvore direita. Se o valor a ser inserido é idêntico ao valor dos dados no nó-raiz, o programa imprime a mensagem "dup" (linha 65) e retorna sem inserir o valor duplicado na árvore. Observe que `insertNode` passa o endereço de `rootPtr` para `insertNodeHelper` (linha 42). Desse modo, ele pode modificar o valor armazenado em `rootPtr` (isto é, o endereço do nó-raiz). Para receber um ponteiro para `rootPtr` (que é também um ponteiro), o primeiro argumento do `insertNodeHelper` é declarado como um ponteiro para um ponteiro para um `TreeNode`.

Cada uma das funções-membro `inOrderTraversal` (linhas 90–94), `preOrderTraversal` (linhas 71–75) e `postOrderTraversal` (linhas 109–113) percorre a árvore e imprime os valores do nó. Para o propósito da próxima discussão, utilizamos a árvore de pesquisa binária da Figura 21.23.

```

1 // Figura 21.22: Fig21_22.cpp
2 // Programa de teste de classe Tree.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::fixed;
7
8 #include <iomanip>
9 using std::setprecision;
10
11 #include "Tree.h" // definição da classe Tree
12
13 int main()
14 {
15 Tree< int > intTree; // cria Tree de valores int
16 int intValue;
17
18 cout << "Enter 10 integer values:\n";
19
20 // insere 10 inteiros em intTree
21 for (int i = 0; i < 10; i++)
22 {
23 cin >> intValue;
24 intTree.insertNode(intValue);
25 } // fim do for
26
27 cout << "\nPreorder traversal\n";
28 intTree.preOrderTraversal();
29
30 cout << "\nInorder traversal\n";

```

**Figura 21.22** Criando e percorrendo uma árvore binária.

(continua)

```

31 intTree.inOrderTraversal();
32
33 cout << "\nPostorder traversal\n";
34 intTree.postOrderTraversal();
35
36 Tree< double > doubleTree; // cria Tree de valores double
37 double doubleValue;
38
39 cout << fixed << setprecision(1)
40 << "\n\nEnter 10 double values:\n";
41
42 // insere 10 doubles em doubleTree
43 for (int j = 0; j < 10; j++)
44 {
45 cin >> doubleValue;
46 doubleTree.insertNode(doubleValue);
47 } // fim do for
48
49 cout << "\nPreorder traversal\n";
50 doubleTree.preOrderTraversal();
51
52 cout << "\nInorder traversal\n";
53 doubleTree.inOrderTraversal();
54
55 cout << "\nPostorder traversal\n";
56 doubleTree.postOrderTraversal();
57
58 cout << endl;
59 return 0;
60 } // fim do main

```

Enter 10 integer values:

**50 25 75 12 33 67 88 6 13 68**

Preorder traversal

50 25 12 6 13 33 75 67 68 88

Inorder traversal

6 12 13 25 33 50 67 68 75 88

Postorder traversal

6 13 12 33 25 68 67 88 75 50

Enter 10 double values:

**39.2 16.5 82.7 3.3 65.2 90.8 1.1 4.4 89.5 92.5**

Preorder traversal

39.2 16.5 3.3 1.1 4.4 82.7 65.2 90.8 89.5 92.5

Inorder traversal

1.1 3.3 4.4 16.5 39.2 65.2 82.7 89.5 90.8 92.5

Postorder traversal

1.1 4.4 3.3 16.5 65.2 89.5 92.5 90.8 82.7 39.2

**Figura 21.22** Criando e percorrendo uma árvore binária.

(continuação)

### Algoritmo do percurso na ordem

A função `inOrderTraversal` invoca a função utilitária `inOrderHelper` para realizar a percurso na ordem da árvore binária. Os passos de um percurso na ordem são:

1. Percorrer a subárvore esquerda com um percurso na ordem. (Isso é realizado pela chamada a `inOrderHelper` na linha 102.)
2. Processar o valor no nó — isto é, imprime o valor de nó (linha 103).
3. Percorrer a subárvore direita com um percurso na ordem. (Isso é realizado pela chamada a `inOrderHelper` na linha 104.)

O valor em um nó não é processado até que os valores em sua subárvore esquerda sejam processados, porque cada chamada para `inOrderHelper` chama imediatamente de novo `inOrderHelper` com o ponteiro para a subárvore esquerda. O percurso na ordem da árvore na Figura 21.23 é

6 13 17 27 33 42 48

Observe que o percurso na ordem de uma árvore de pesquisa binária imprime os valores de nó na ordem crescente. O processo de criar uma árvore de pesquisa binária realmente classifica os dados — portanto, esse processo é chamado **classificação de árvore binária**.

### Algoritmo de percurso pré-ordem

A função `preOrderTraversal` invoca a função utilitária `preOrderHelper` para realizar o percurso pré-ordem da árvore binária. Os passos de um percurso pré-ordem são:

1. Processar o valor no nó (linha 83).
2. Percorrer a subárvore esquerda com um percurso pré-ordem. (Isso é realizado pela chamada a `preOrderHelper` na linha 84.)
3. Percorrer a subárvore direita com um percurso pré-ordem. (Isso é realizado pela chamada a `preOrderHelper` na linha 85.)

O valor em cada nó é processado quando o nó é visitado. Depois que o valor em um dado nó é processado, os valores na subárvore esquerda são processados. Então os valores na subárvore direita são processados. O percurso pré-ordem da árvore na Figura 21.23 é

27 13 6 17 42 33 48

### Algoritmo de percurso pós-ordem

A função `postOrderTraversal` invoca a função utilitária `postOrderHelper` para realizar a percurso pós-ordem da árvore binária. Os passos de um percurso pós-ordem são:

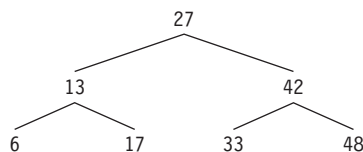
1. Percorrer a subárvore esquerda com um percurso pós-ordem. (Isso é realizado pela chamada a `postOrderHelper` na linha 122.)
2. Percorrer a subárvore direita com um percurso pós-ordem. (Isso é realizado pela chamada a `postOrderHelper` na linha 123.)
3. Processar o valor no nó (linha 124).

O valor em cada nó não é impresso até os valores de seus filhos serem impressos. A `postOrderTraversal` da árvore na Figura 21.23 é

6 17 13 33 48 42 27

### Eliminação de duplicatas

A árvore de pesquisa binária facilita a **eliminação de duplicatas**. À medida que a árvore é criada, uma tentativa de inserir um valor duplicado será reconhecida, porque uma duplicata seguirá as mesmas decisões ‘vá para a esquerda’ ou ‘vá para a direita’ em cada comparação da mesma forma que o valor original faz ao ser inserido na árvore. Portanto, a duplicata será finalmente comparada com um nó que contém o mesmo valor. O valor duplicado pode ser descartado nesse ponto.



**Figura 21.23** Uma árvore de pesquisa binária.



Procurar em uma árvore binária um valor que corresponda a um valor-chave também é rápido. Se a árvore é equilibrada, então cada desvio contém cerca da metade do número de nós na árvore. Cada comparação de um nó com a chave de pesquisa elimina metade dos nós. Isso é chamado de algoritmo  $O(\log n)$  (a notação  $O$  é discutida no Capítulo 20). Portanto, uma árvore de pesquisa binária com  $n$  elementos requeriria um máximo de  $\log_2 n$  comparações para localizar uma correspondência ou determinar que não há nenhuma correspondência. Isso significa, por exemplo, que, ao pesquisar em uma árvore (equilibrada) de pesquisa binária de 1.000 elementos, não é preciso fazer mais que 10 comparações, porque  $2^{10} > 1.000$ . Ao pesquisar em uma árvore (equilibrada) de pesquisa binária de 1.000.000 elementos, não mais que 20 comparações são necessárias, porque  $2^{20} > 1.000.000$ .

### Visão geral dos exercícios de árvore binária

Nos exercícios, os algoritmos são apresentados para várias outras operações de árvore binária, como excluir um item de uma árvore binária, imprimir uma árvore binária em um formato de árvore bidimensional e realizar um percurso na ordem de nível de uma árvore binária. O percurso na ordem de nível de uma árvore binária percorre os nós da árvore linha por linha iniciando no nível do nó-raiz. Em cada nível da árvore, os nós são percorridos da esquerda para a direita. Outros exercícios de árvore binária incluem permitir que uma árvore de pesquisa binária contenha valores duplicados, insira valores de string em uma árvore binária e determine quantos níveis estão contidos em uma árvore binária.

## 21.8 Síntese

Neste capítulo, você aprendeu que as listas vinculadas são coleções de itens de dados ‘vinculados em uma cadeia’. Também aprendeu que um programa pode realizar inserções e exclusões em qualquer lugar de uma lista vinculada (embora nossa implementação tenha executado somente inserções e exclusões nos finais da lista). Demonstramos que as estruturas de dados da pilha e de fila são versões restritas de listas. Quanto às pilhas, vimos que as inserções e exclusões só podem ser feitas no topo. Quanto às filas que representam seqüências de espera, você viu que inserções são feitas na cauda (na parte de trás) e as exclusões são feitas na cabeça (na parte da frente). Apresentamos também a estrutura de dados de árvore binária. Você viu uma árvore de pesquisa binária que facilitou a pesquisa e classificação de dados em alta velocidade e a eliminação eficiente de duplicatas. Por todo o capítulo, você aprendeu a criar essas estruturas de dados para facilitar o reúso (como templates) e a manutenção. No próximo capítulo, introduziremos as `structs`, que são semelhantes às classes, e discutiremos a manipulação de bits, caracteres e strings no estilo do C.

### Resumo

- As estruturas de dados dinâmicas crescem e encolhem durante a execução.
- As listas vinculadas são coleções de itens de dados ‘vinculados em uma cadeia’ — inserções e exclusões são feitas em qualquer lugar de uma lista vinculada.
- As pilhas são importantes em compiladores e sistemas operacionais: as inserções e exclusões são feitas somente no final de uma pilha — sua parte superior.
- As filas representam filas de espera; as inserções são feitas na parte de trás (também referida como cauda) de uma fila e as remoções são feitas na parte da frente da fila (também referida como cabeça).
- As árvores binárias facilitam a pesquisa e a classificação de dados em alta velocidade, a eliminação eficiente de itens de dados duplicados, a representação de diretórios de sistema de arquivos e a compilação de expressões em linguagem de máquina.
- Uma classe auto-referencial contém um membro ponteiro que aponta para um objeto de classe do mesmo tipo de classe.
- Os objetos de classe auto-referencial podem ser vinculados para formar estruturas de dados úteis como listas, filas, pilhas e árvores.
- O limite para alocação dinâmica de memória pode ser tão grande quanto a quantidade de memória física disponível no computador ou a quantidade de memória virtual disponível em um sistema de memória virtual.
- Uma lista vinculada é uma coleção linear de objetos de classe auto-referencial, chamados nós, conectados por vínculos de ponteiro — daí, o termo lista ‘vinculada’.
- Uma lista vinculada é acessada via ponteiro ao primeiro nó da lista. Cada nó subsequente é acessado via membro ponteiro de vínculo armazenado no nó anterior.
- As listas vinculadas, pilhas e filas são estruturas de dados lineares. As árvores são estruturas de dados não-lineares.
- Uma lista vinculada é apropriada quando o número de elementos de dados a ser representado em um tempo é imprevisível.
- As listas vinculadas são dinâmicas, portanto o comprimento de uma lista pode aumentar ou diminuir conforme necessário.
- Uma lista simplesmente vinculada inicia com um ponteiro para o primeiro nó e cada nó contém um ponteiro para o próximo nó ‘na seqüência’.
- Uma lista circular simplesmente vinculada inicia com um ponteiro para o primeiro nó e cada nó contém um ponteiro para o próximo nó. O ‘último nó’ não contém um ponteiro nulo; em vez disso, o ponteiro no último nó aponta de volta para o primeiro nó, fechando, assim, o ‘círculo’.

- Uma lista duplamente vinculada permite tanto percursos para a frente como para trás.
- Uma lista duplamente vinculada é frequentemente implementada com dois ‘ponteiros iniciais’ — um que aponta para o primeiro elemento da lista a fim de permitir percurso de frente para trás na lista, e outro que aponta para o último elemento a fim de permitir percurso de trás para a frente. Cada nó tem um ponteiro para a frente para o próximo nó na lista na direção para a frente, e um ponteiro para trás para o próximo nó na direção para trás.
- Em uma lista circular duplamente vinculada, o ponteiro para a frente do último nó aponta para o primeiro nó, e o ponteiro para trás do primeiro nó aponta para o último nó, fechando, assim, o ‘círculo’.
- Uma estrutura de dados de pilha permite que nós sejam adicionados à pilha e removidos da pilha somente na parte superior.
- Uma pilha é referida como uma estrutura de dados último a entrar, primeiro a sair (*last-in, first-out* – LIFO).
- As principais funções-membro utilizadas para manipular uma pilha são push e pop. A função push insere um novo nó na parte superior da pilha. A função pop remove um nó da parte superior da pilha.
- Uma fila é semelhante a uma fila de caixa em um supermercado — a primeira pessoa na fila é atendida primeiro, e outros clientes entram no fim da fila e esperam o atendimento.
- Os nós da fila são removidos apenas do início (ou cabeça) da fila e são inseridos somente no final (ou cauda) da fila.
- Uma fila é referida como uma estrutura de dados primeiro a entrar, primeiro a sair (*first-in, first-out* – FIFO). As operações de inserção e remoção são conhecidas como enqueue e dequeue.
- As árvores binárias são árvores cujos nós contêm dois vínculos (nenhum, um ou ambos podem ser nulos).
- O nó-raiz é o primeiro nó em uma árvore.
- Cada vínculo no nó-raiz referencia um filho. O filho esquerdo é o nó-raiz da subárvore esquerda e o filho direito é o nó-raiz da subárvore direita.
- Os filhos de um nó individual são chamados irmãos. Um nó sem filhos é chamado de nó-folha.
- Uma árvore de pesquisa binária (sem valores duplicados de nó) tem a característica de que os valores em qualquer subárvore esquerda são menores que o valor em seu nó-pai e os valores em qualquer subárvore direita são maiores que o valor em seu nó-pai.
- Um nó pode apenas ser inserido como um nó-folha em uma árvore de pesquisa binária.
- Um percurso na ordem de uma árvore binária percorre a subárvore esquerda na ordem, processa o valor no nó-raiz e, então, percorre a subárvore direita na ordem. O valor em um nó não é processado até os valores em sua subárvore esquerda serem processados.
- Um percurso pré-ordem processa o valor no nó-raiz, percorre a subárvore esquerda na pré-ordem e, então, percorre a subárvore direita na pré-ordem. O valor em cada nó é processado quando o nó é encontrado.
- Um percurso pós-ordem percorre a subárvore esquerda na pós-ordem, percorre a subárvore direita na pós-ordem e, então, processa o valor no nó-raiz. O valor em cada nó não é processado até que os valores em suas duas subárvores sejam processados.
- A árvore de pesquisa binária facilita a eliminação de duplicatas. Durante a criação da árvore, uma tentativa de inserir um valor duplicado será reconhecida e o valor duplicado pode ser descartado.
- O percurso na ordem de nível de uma árvore binária percorre os nós da árvore linha por linha iniciando no nível do nó-raiz. Em cada nível da árvore, os nós são percorridos da esquerda para a direita.

## Terminologia

árvore binária	filho esquerdo	percurso na ordem de uma árvore binária
árvore de pesquisa binária	final de uma fila	percurso na ordem do nível
cabeça de uma fila	inserir um nó	percurso pós-ordem de uma árvore binária
classificação árvore binária	irmãos	percurso pré-ordem de uma árvore binária
delegação	lista circular duplamente vinculada	pilha
dequeue	lista circular simplesmente vinculada	pop
eliminação de duplicatas	lista duplamente vinculada	primeiro a entrar, primeiro a sair ( <i>first-in, first-out</i> – FIFO)
enqueue	lista simplesmente vinculada	push
estrutura auto-referencial	lista vinculada	spooler
estrutura de dados	nó	spooling de impressão
estrutura de dados linear	nó-filho	subárvore direita
estrutura de dados não-linear	nó-folha	subárvore esquerda
estruturas de dados dinâmicas	nó-pai	último a entrar, primeiro a sair ( <i>last-in, first-out</i> – LIFO)
fila	nó-raiz	vínculo ( <i>link</i> )
filho direito	parte superior de uma pilha	vínculo de ponteiro

## Exercícios de revisão

**21.1** Preencha as lacunas em cada uma das seguintes sentenças:

- Uma classe auto-\_\_\_\_\_ é utilizada para formar estruturas de dados dinâmicas que podem crescer e encolher em tempo de execução.
- O operador \_\_\_\_\_ é utilizado para alocar memória dinamicamente e construir um objeto; esse operador retorna um ponteiro para o objeto.
- Um(a) \_\_\_\_\_ é uma versão restrita de uma lista vinculada em que os nós podem ser inseridos e excluídos somente a partir do início da lista e valores de nó são retornados na ordem último a entrar, primeiro a sair.
- Uma função que não altera uma lista vinculada, mas examina a lista para determinar se ela está vazia, é um exemplo de uma função \_\_\_\_\_.
- Uma fila é referida como uma estrutura de dados \_\_\_\_\_ porque os primeiros nós inseridos são os primeiros nós removidos.
- O ponteiro para o próximo nó em uma lista vinculada é referido como \_\_\_\_\_.
- O operador \_\_\_\_\_ é utilizado para destruir um objeto e liberar memória dinamicamente alocada.
- Um(a) \_\_\_\_\_ é uma versão restrita de uma lista vinculada em que os nós podem ser inseridos apenas no final da lista e excluídos apenas do início da lista.
- Um(a) \_\_\_\_\_ é uma estrutura de dados bidimensional não-linear que contém nós com dois ou mais vínculos.
- Uma pilha é referida como uma estrutura de dados \_\_\_\_\_, porque o último nó inserido é o primeiro nó removido.
- Os nós de uma árvore \_\_\_\_\_ contém dois membros de vínculo.
- O primeiro nó de uma árvore é o nó- \_\_\_\_\_.
- Cada vínculo em um nó de árvore aponta para um \_\_\_\_\_ ou \_\_\_\_\_ desse nó.
- Um nó de árvore que não tem filhos é chamado de nó- \_\_\_\_\_.
- Os quatro algoritmos de percursos que mencionamos no texto para árvores de pesquisa binária são \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_ e \_\_\_\_\_.

**21.2** Quais são as diferenças entre uma lista vinculada e uma pilha?

**21.3** Quais são as diferenças entre uma pilha e uma fila?

**21.4** Talvez um título mais apropriado para este capítulo fosse “Estruturas de dados reutilizáveis”. Comente como cada uma das seguintes entidades ou conceitos contribuem para a capacidade de reutilização das estruturas de dados:

- classes
- templates de classe
- herança
- herança *private*
- composição

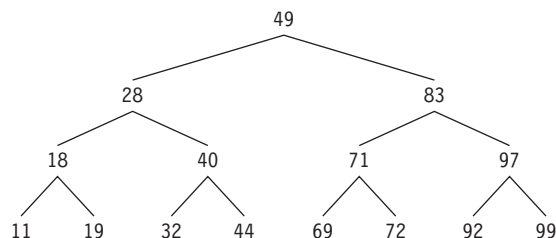
**21.5** Forneça manualmente os percursos na ordem, pré-ordem e pós-ordem da árvore de pesquisa binária da Figura 21.24.

## Respostas dos exercícios de revisão

**21.1** a) referencial. b) *new*. c) pilha. d) predicado. e) primeiro a entrar, primeiro a sair (*first-in, first-out* – FIFO). f) vínculo (*link*). g) *delete*. h) fila. i) árvore. j) último a entrar, primeiro a sair (*last-in, first-out* – LIFO). k) binária. l) raiz. m) filho ou subárvore. n) folha. o) na ordem, pré-ordem, pós-ordem e na ordem do nível.

**21.2** É possível inserir um nó em qualquer lugar em uma lista vinculada e remover um nó de qualquer lugar em uma lista vinculada. Os nós em uma pilha só podem ser inseridos na parte superior da pilha e só podem ser removidos da parte superior de uma pilha.

**21.3** Uma estrutura de dados de fila permite que os nós sejam removidos somente da parte superior da fila e inseridos somente na parte inferior da fila. Uma fila é referida como uma estrutura de dados primeiro a entrar, primeiro a sair (*first-in, first-out* – FIFO). Uma estrutura de



**Figura 21.24** Um árvore de pesquisa binária de 15 nós.

dados de pilha permite que nós sejam adicionados à pilha e removidos da pilha somente na parte superior. Uma pilha é referida como uma estrutura de dados último a entrar, primeiro a sair (*last-in, first-out* – LIFO).

- 21.4**
- a) As classes nos permitem instanciar quantos objetos de estrutura de dados de certo tipo (isto é, classe) quisermos.
  - b) Os templates de classe nos permitem instanciar classes relacionadas, cada uma com base em diferentes parâmetros de tipo — então podemos gerar quantos objetos de cada classe de template quisermos.
  - c) A herança nos permite reutilizar o código de uma classe básica em uma classe derivada, de modo que a estrutura de dados de classe derivada também seja uma estrutura de dados de classe básica (isto é, com a herança pública).
  - d) Herança `private` permite reutilizar partes do código de uma classe básica para formar uma estrutura de dados de classe derivada; como a herança é `private`, todas as funções-membro `public` de classe básica tornam-se `private` na classe derivada. Isso nos permite impedir que clientes da estrutura de dados de classe derivada acessem as funções-membro da classe básica que não se aplicam à classe derivada.
  - e) A composição nos permite reutilizar código tornando uma estrutura de dados de um objeto de classe um membro de uma classe composta; se tornarmos o objeto de classe um membro `private` da classe composta, então as funções-membro públicas do objeto de classe não são disponíveis por meio da interface do objeto composto.

- 21.5** O percurso na ordem é

11 18 19 28 32 40 44 49 69 71 72 83 92 97 99

O percurso pré-ordem é

49 28 18 11 19 40 32 44 83 71 69 72 97 92 99

O percurso pós-ordem é

11 19 18 32 44 40 28 69 72 71 92 99 97 83 49

## Exercícios

- 21.6** Escreva um programa que concatena dois objetos de lista vinculada de caracteres. O programa deve incluir a função `concatenate`, que aceita referências a ambos os objetos da lista como argumentos e concatena a segunda com a primeira lista.
- 21.7** Escreva um programa que mescla dois objetos ordenados de lista de inteiros em um único objeto ordenado de lista de inteiros. A função `merge` deve receber referências a cada um dos objetos de lista a ser mesclado e referenciar o objeto lista em que os elementos mesclados serão colocados.
- 21.8** Escreva um programa que insere 25 inteiros aleatórios de 0 a 100 na ordem em um objeto lista vinculada. O programa deve calcular a soma dos elementos e a média de ponto flutuante dos elementos.
- 21.9** Escreva um programa que cria um objeto lista vinculada de 10 caracteres e cria um segundo objeto lista contendo uma cópia da primeira lista, mas na ordem inversa.
- 21.10** Escreva um programa que insere uma linha de texto e utiliza um objeto pilha para imprimir a linha invertida.
- 21.11** Escreva um programa que utiliza um objeto pilha para determinar se uma string é um palíndromo (isto é, a string é grafada identicamente da esquerda para a direita e da direita para a esquerda). O programa deve ignorar espaços e pontuação.
- 21.12** As pilhas são utilizadas por compiladores para ajudar no processo de avaliação de expressões e na geração de código de linguagem de máquina. Neste e no próximo exercício, investigamos como os compiladores avaliam expressões aritméticas que consistem apenas de constantes, operadores e parênteses.

Humanos geralmente escrevem expressões como  $3 + 4 \times 7 / 9$  em que o operador (+ ou / aqui) é escrito entre seus operandos — isso é chamado **notação infixa**. Os computadores ‘preferem’ **notação pós-fixa**, em que o operador é escrito à direita de seus dois operandos. As expressões infixas precedentes apareceriam na notação pós-fixa como  $3\ 4\ +\ 7\ 9\ /\$ , respectivamente.

Para avaliar uma expressão infixa complexa, um compilador primeiro converteria a expressão em notação pós-fixa e então avaliaria a versão da expressão pós-fixa. Cada um desses algoritmos requer apenas uma única passagem da esquerda para a direita pela expressão. Cada algoritmo utiliza um objeto pilha em suporte à sua operação, e em cada algoritmo a pilha é utilizada para um propósito diferente.

Neste exercício, você escreverá uma versão C++ do algoritmo de conversão de infixo para pós-fixo. No próximo exercício, você escreverá uma versão C++ do algoritmo de avaliação de expressão pós-fixa. Mais adiante no capítulo, você descobrirá que o código que você escreve neste exercício pode ajudá-lo a implementar um compilador funcional completo.

Escreva um programa que converte uma expressão aritmética infixa comum (suponha que uma expressão válida é inserida) com inteiros de único dígito como

$(6 + 2) * 5 - 8 / 4$

para uma expressão pós-fixa. A versão pós-fixa da expressão infixa precedente é

$6\ 2\ +\ 5\ *\ 8\ 4\ /\ -$

O programa deve ler a expressão no array de caracteres `infix` e utilizar as versões modificadas das funções de pilha implementadas neste capítulo para ajudar a criar a expressão pós-fixa no array de caracteres `postfix`. O algoritmo para criar uma expressão pós-fixa é o seguinte:

- 1) Insira um parêntese esquerdo '(' na pilha.
- 2) Acrescente um parêntese direito ')' ao final de `infix`.
- 3) Enquanto a pilha não estiver vazia, leia `infix` da esquerda para a direita e faça o seguinte:
  - Se o caractere atual no `infix` for um dígito, copie-o para o próximo elemento de `postfix`.
  - Se o caractere atual em `infix` for um parêntese esquerdo, adicione-o à pilha.
  - Se o caractere atual no `infix` for um operador,
    - Remova os operadores (se houver algum) a partir da parte superior da pilha, embora eles tenham precedência igual ou maior que a do operador atual e insira os operadores removidos em `postfix`.
    - Adicione o caractere atual a `infix` na pilha.
  - Se o caractere atual no `infix` for um parêntese direito,
    - Remova os operadores a partir da parte superior da pilha e os insira em `postfix` até que um parêntese esquerdo esteja na parte superior da pilha.
    - Remova (e descarte) o parêntese esquerdo da pilha.

As seguintes operações aritméticas são permitidas em uma expressão:

- + adição
- subtração
- \* multiplicação
- / divisão
- ^ exponenciação
- % módulo

[Nota: Para o propósito deste exercício, supomos a associatividade da esquerda para a direita para todos os operadores.] A pilha deve ser mantida com os nós de pilha, cada nó contendo um membro de dados e um ponteiro para o próximo nó de pilha.

Algumas capacidades funcionais que você pode querer fornecer são:

- a) A função `convertToPostfix`, que converte a expressão infixa em notação pós-fixa.
- b) A função `isOperator`, que determina se `c` é um operador.
- c) A função `precedence`, que determina se a precedência de `operator1` é menor que, igual a ou maior que a precedência de `operator2` (a função retorna -1, 0 e 1, respectivamente).
- d) A função `push`, que insere um valor na pilha.
- e) A função `pop`, que remove um valor da pilha.
- f) A função `stackTop`, que retorna o valor superior da pilha sem remover a pilha.
- g) A função `isEmpty`, que determina se a pilha está vazia.
- h) A função `printStack`, que imprime a pilha.

## 21.13 Escreva um programa que avalia uma expressão pós-fixa (pressuponha que ela seja válida) como

6 2 + 5 \* 8 4 / -

O programa deve ler uma expressão pós-fixa consistindo em dígitos e operadores em um array de caracteres. Utilizando versões modificadas das funções de pilha implementadas anteriormente neste capítulo, o programa deve varrer a expressão e avaliá-la. O algoritmo é como segue:

- 1) Acrescente o caractere nulo ('\0') ao fim da expressão pós-fixa. Quando o caractere nulo é encontrado, não é necessário mais nenhum processamento.
- 2) Enquanto '\0' não foi encontrado, leia a expressão da esquerda para a direita.
  - Se o caractere atual for um dígito,
    - Insira seu valor inteiro na pilha (o valor de inteiro de um caractere de dígito é seu valor no conjunto de caracteres do computador menos o valor de '0' no conjunto de caracteres do computador).
  - Caso contrário, se o caractere atual for um *operador*,
    - Remova os dois elementos superiores da pilha para variáveis `x` e `y`.
    - Calcule `y operador x`.
    - Adicione o resultado do cálculo à pilha.
- 3) Quando o caractere nulo for encontrado na expressão, remova o valor superior da pilha. Esse é o resultado da expressão pós-fixa.

[Nota: No Passo 2 citado, se o operador for '/', a parte superior da pilha é 2 e o próximo elemento na pilha é 8, então remova 2 em `x`, remova 8 em `y`, avalie `8 / 2` e insira o resultado, 4, de volta na pilha. Essa nota também se aplica ao operador '-'.] As operações aritméticas permitidas em uma expressão são

- + adição
- subtração
- \* multiplicação
- / divisão
- ^ exponenciação
- % módulo

[Nota: Para o propósito deste exercício, supomos a associatividade da esquerda para a direita para todos os operadores.] A pilha deve ser mantida com nós de pilha que contêm um membro de dados `int` e um ponteiro para o próximo nó da pilha. Você pode querer fornecer as seguintes capacidades funcionais:

- a) A função `evaluatePostfixExpression`, que avalia a expressão pós-fixa.
- b) A função `calculate`, que avalia a expressão `op1 operator op2`.
- c) A função `push`, que insere um valor na pilha.
- d) A função `pop`, que remove um valor da pilha.
- e) A função `isEmpty`, que determina se a pilha está vazia.
- f) A função `printStack`, que imprime a pilha.

**21.14** Modifique o programa avaliador de pós-fixado do Exercício 21.13 de modo que ele possa processar os operandos de inteiro maiores que 9.

**21.15** (*Simulação de supermercado*) Escreva um programa que simula uma fila de caixa em um supermercado. A fila é um objeto fila. Os clientes (isto é, objetos-cliente) chegam em intervalos de inteiro aleatório de 1–4 minutos. Além disso, cada cliente é atendido em intervalos de inteiro aleatório de 1–4 minutos. Obviamente, as taxas precisam ser equilibradas. Se a taxa média de chegada for maior que a taxa média de atendimento, a fila crescerá infinitamente. Mesmo com taxas ‘equilibradas’, a aleatoriedade ainda pode provocar filas longas. Execute a simulação de supermercado para um dia de 12 horas (720 minutos) utilizando o seguinte algoritmo:

- 1) Escolha um inteiro aleatório entre 1 e 4 para determinar o minuto em que o primeiro cliente chega.
- 2) Na hora de chegada do primeiro cliente:

Determine o tempo de atendimento do serviço ao cliente (inteiro aleatório de 1 a 4).

Comece atendendo o cliente.

Agende a hora de chegada do próximo cliente (inteiro aleatório de 1 a 4 adicionado à hora atual).

- 3) Para cada minuto do dia:

Se o próximo cliente chegar,

Expresse isso.

Enfileire o cliente.

Agende a hora de chegada do próximo cliente.

Se o serviço foi completado para o último cliente,

Expresse isso.

Desenfileire o próximo cliente a ser atendido.

Determine o tempo de conclusão do serviço de atendimento ao cliente (inteiro aleatório de 1 a 4 adicionado à hora atual).

Agora execute sua simulação para 720 minutos e responda a cada uma das seguintes questões:

- a) Qual é o número máximo de clientes na fila a qualquer hora?
- b) Qual é a espera mais longa que qualquer cliente experimenta?
- c) O que acontece se o intervalo de chegada é mudado de 1–4 minutos para 1–3 minutos?

**21.16** Modifique o programa das figuras 21.20–21.22 para permitir que o objeto árvore binária contenha duplicatas.

**21.17** Escreva um programa baseado nas figuras 21.20–21.22 que insira uma linha de texto, tokenize a frase em palavras separadas (você pode querer utilizar a função de biblioteca `strtok`), insira as palavras em uma árvore de pesquisa binária e imprima os percursos na ordem, pré-ordem e pós-ordem da árvore. Utilize uma abordagem OOP.

**21.18** Neste capítulo, vimos que a eliminação de duplicatas é simples e direta quando se cria uma árvore de pesquisa binária. Descreva como você realizaria a eliminação de duplicatas utilizando apenas um array unidimensional. Compare o desempenho da eliminação de duplicatas baseada em array com o desempenho da eliminação de duplicatas baseada na pesquisa de árvore binária.

**21.19** Escreva uma função `depth` que recebe uma árvore binária e determine quantos níveis ela tem.

**21.20** (*Imprimir recursivamente uma lista de trás para a frente*) Escreva uma função-membro `printListBackward` que recursivamente gera saída dos itens em um objeto lista vinculada na ordem inversa. Escreva um programa de teste que cria uma lista classificada de inteiros e imprime a lista em ordem inversa.

**21.21** (*Pesquisar recursivamente uma lista*) Escreva uma função-membro `searchList` que pesquisa recursivamente um valor especificado em um objeto lista vinculada. A função deve retornar um ponteiro para o valor se ele for localizado; caso contrário, nulo deve ser retornado.



Utilize sua função em um programa de teste que cria uma lista de inteiros. O programa deve solicitar ao usuário um valor para localizar na lista.

**21.22** (*Exclusão de árvore binária*) Neste exercício, discutimos a exclusão de itens de árvores de pesquisa binária. O algoritmo de exclusão não é tão simples e direto quanto o algoritmo de inserção. Há três casos que são encontrados ao excluir um item — o item está contido em um nó-folha (isto é, não tem filhos), o item está contido em um nó que tem um filho ou o item está contido em um nó que tem dois filhos.

Se o item a ser excluído está contido em um nó-folha, o nó é excluído e o ponteiro no nó-pai é configurado como nulo.

Se o item a ser excluído está contido em um nó com um filho, o ponteiro no nó-pai é configurado para apontar para o nó-filho, e o nó contendo o item de dados é excluído. Isso faz com que o nó-filho tome o lugar do nó excluído na árvore.

O último caso é o mais difícil. Quando um nó com dois filhos é excluído, outro nó na árvore deve tomar seu lugar. Entretanto, o ponteiro no nó-pai não pode ser atribuído para apontar para um dos filhos do nó a ser excluído. Na maioria dos casos, a árvore de pesquisa binária resultante não obedeceria à seguinte característica das árvores de pesquisa binária (sem valores duplicados): *os valores em qualquer subárvore esquerda são menores que o valor no nó-pai, e os valores em qualquer subárvore direita são maiores que o valor no nó-pai.*

Qual é o nó utilizado como um *nó substituto* para manter essa característica? O nó contendo o maior valor na árvore menor que o valor no nó sendo excluído, ou o nó contendo o menor valor na árvore maior que o valor no nó sendo excluído? Vamos considerar o nó com o menor valor. Em uma árvore de pesquisa binária, o maior valor menor que um valor do pai encontra-se na subárvore esquerda do nó-pai e seguramente estará contido no nó mais à direita da subárvore. Esse nó é encontrado descendendo a subárvore esquerda pela direita até que o ponteiro para o filho direito do nó atual seja nulo. Agora estamos apontando para o nó substituto que é um nó-folha ou um nó com um filho à sua esquerda. Se o nó substituto for um nó-folha, os passos para realizar a exclusão são os seguintes:

- 1) Armazene o ponteiro para o nó a ser excluído em uma variável de ponteiro temporária (esse ponteiro é utilizado para excluir a memória dinamicamente alocada).
- 2) Configure o ponteiro no pai do nó sendo excluído para apontar para o nó substituto.
- 3) Configure o ponteiro no pai do nó substituto como nulo.
- 4) Configure o ponteiro como a subárvore direita no nó substituto para apontar para a subárvore direita do nó a ser excluído.
- 5) Exclua o nó para o qual a variável ponteiro temporária aponta.

Os passos de exclusão para um nó substituto com um filho esquerdo são semelhantes àqueles para um nó substituto sem filhos, mas o algoritmo também deve mover o filho para a posição do nó substituto na árvore. Se o nó substituto for um nó com um filho esquerdo, os passos para realizar a exclusão são como segue:

- 1) Armazene o ponteiro para o nó a ser excluído em uma variável ponteiro temporária.
- 2) Configure o ponteiro no pai do nó sendo excluído para apontar para o nó substituto.
- 3) Configure o ponteiro no pai do nó substituto para apontar para o filho esquerdo do nó substituto.
- 4) Configure o ponteiro como a subárvore direita no nó substituto para apontar para a subárvore direita do nó a ser excluído.
- 5) Exclua o nó para o qual a variável ponteiro temporária aponta.

Escreva a função-membro `deleteNode`, que aceita como seus argumentos um ponteiro para o nó-raiz do objeto-árvore e o valor a ser excluído. A função deve localizar na árvore o nó contendo o valor a ser excluído e utilizar os algoritmos discutidos aqui para excluir o nó. A função deve imprimir uma mensagem que indica se o valor foi excluído. Modifique o programa das figuras 21.20–21.22 para utilizar essa função. Depois de excluir um item, chame as funções de percurso `inOrder`, `preOrder` e `postOrder` para confirmar se a operação de exclusão foi realizada corretamente.

**21.23** (*Pesquisa de árvore binária*) Escreva a função-membro `binaryTreeSearch`, que tenta localizar um valor especificado em um objeto-árvore de pesquisa binária. A função deve aceitar como argumentos um ponteiro para o nó-raiz da árvore binária e uma chave de pesquisa a ser localizada. Se o nó contendo a chave de pesquisa for localizado, a função deve retornar um ponteiro para aquele nó; caso contrário, a função deve retornar um ponteiro nulo.

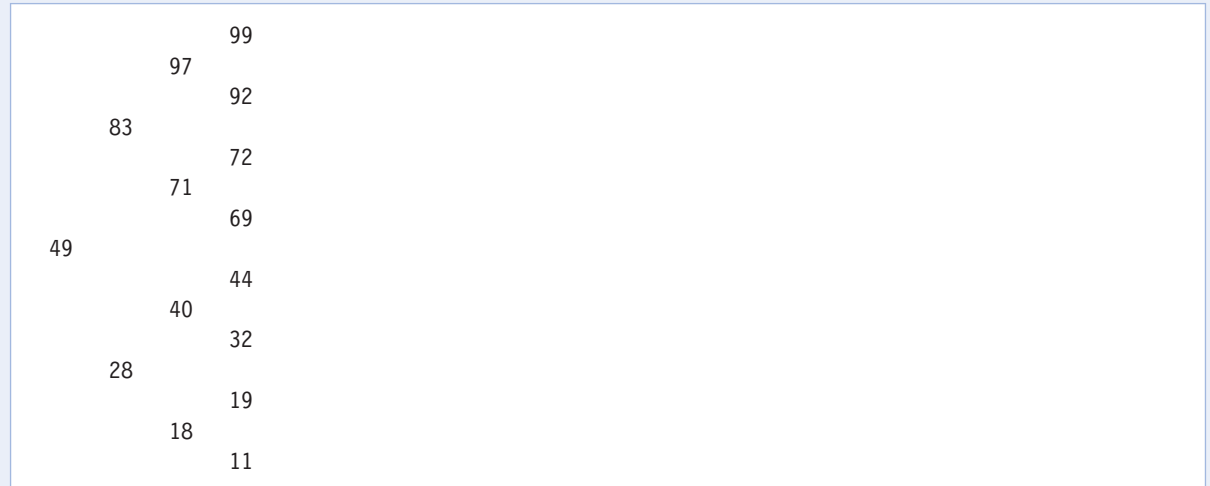
**21.24** (*Percurso de árvore binária na ordem de nível*) O programa das figuras 21.20–21.22 ilustrou três métodos recursivos de percorrer uma árvore binária — os percursos na ordem, pré-ordem e pós-ordem. Este exercício apresenta o percurso *na ordem de nível* de uma árvore binária, no qual os valores de nó são impressos nível por nível iniciando no nível do nó-raiz. Os nós em cada nível são impressos da esquerda para a direita. O percurso na ordem de nível não é um algoritmo recursivo. Ela utiliza um objeto fila para controlar a saída dos nós. O algoritmo é como segue:

- 1) Insira o nó-raiz na fila.
- 2) Enquanto houver nós esquerdos na fila,
  - Obtenha o próximo nó na fila.
  - Imprima o valor do nó.
  - Se o ponteiro para o filho esquerdo do nó não for nulo,
    - Insira o nó-filho esquerdo na fila.
  - Se o ponteiro para o filho direito do nó não for nulo,
    - Insira o nó-filho direito na fila.



Escreva a função-membro `levelOrder` para realizar um percurso na ordem de nível de um objeto árvore binária. Modifique o programa das figuras 21.20–21.22 para utilizar essa função. [Nota: Você também precisará modificar e incorporar as funções de processamento fila da Figura 21.16 neste programa.]

**21.25** (*Imprimindo árvores*) Escreva uma função-membro recursiva `outputTree` para exibir um objeto árvore binária na tela. A função deve gerar saída da árvore linha por linha com a parte superior da árvore na parte esquerda da tela e a parte inferior da árvore em direção à parte direita da tela. Cada linha é enviada para a saída verticalmente. Por exemplo, a saída da árvore binária ilustrada na Figura 21.24 é realizada como mostrado a seguir:



Observe que o nó mais à direita da folha aparece na parte superior da saída na coluna mais à direita e o nó-raiz aparece à esquerda da saída. Cada coluna de saída inicia cinco espaços à direita da coluna anterior. A função `outputTree` deve receber um argumento `totalSpaces` representando o número de espaços que precedem o valor a ser enviado para a saída (essa variável deve iniciar em zero de modo que o nó-raiz seja enviado para a saída na parte esquerda da tela). A função utiliza um percurso na ordem modificado para gerar a saída da árvore — ela inicia no nó mais à direita na árvore e segue para a esquerda. O algoritmo é como segue:

```

Enquanto o ponteiro para o nó atual não é nulo,
 Chame recursivamente outputTree com a subárvore direita do nó atual e
 totalSpaces + 5
 Utilize uma estrutura for para contar de 1 a totalSpaces e gerar saída de espaços.
 Envie para a saída o valor no nó atual.
 Configure o ponteiro como o nó atual para apontar para a subárvore esquerda do nó atual.
 Incremente totalSpaces por 5.

```

## Seção especial: construindo seu próprio compilador

Nos exercícios 8.18 e 8.19, introduzimos a Simpletron Machine Language (SML) e você implementou um simulador de computador Simpletron para executar programas escritos em SML. Nesta seção, construímos um compilador que converte programas escritos em uma linguagem de programação de alto nível em SML. Esta seção ‘amarra’ o processo de programação inteiro. Você irá escrever os programas nessa nova linguagem de alto nível, compilar esses programas no compilador construído e executá-los no simulador que construiu no Exercício 8.19. Você deve se esforçar o máximo possível para implementar seu compilador de uma maneira orientada a objetos.

**21.26** (*A linguagem Simple*) Antes de iniciarmos a construção do compilador, discutimos uma linguagem simples, mas ainda poderosa e de alto nível, semelhante a versões anteriores da popular linguagem BASIC. Chamamos essa linguagem de Simple. Cada *instrução* (*statement*) Simple consiste em um *número de linha* e uma *instrução* (*instruction*) Simple propriamente dita. Os números da linha devem aparecer em ordem crescente. Cada instrução inicia com um dos seguintes *comandos* Simple: `rem`, `input`, `let`, `print`, `goto`, `if...goto` e `end` (ver Figura 21.25). Todos os comandos, exceto `end`, podem ser utilizados repetidamente. O Simple avalia apenas as expressões de inteiro que utilizam os operadores `+`, `-`, `*` e `/`. Esses operadores têm a mesma precedência em C++. Os parênteses podem ser utilizados para alterar a ordem de avaliação de uma expressão.

Nosso compilador de Simple reconhece apenas letras minúsculas. Todos os caracteres em um arquivo Simple devem estar em letras minúsculas (letras maiúsculas resultam em um erro de sintaxe, a menos que apareçam em uma instrução `rem`, caso em que são ignoradas). Um nome de variável tem uma única letra. A linguagem Simple não permite nomes de variáveis descritivos, portanto as variáveis devem ser explicadas em observações para indicar sua utilização em um programa. O Simple utiliza apenas variáveis de inteiro. O Simple não tem declarações de variável — a mera menção a um nome de variável em um programa faz com que a variável seja declarada e inicializada.

Comando	Instrução de exemplo	Descrição
rem	50 rem this is a remark	O texto que se segue a rem é para fins de documentação e é ignorado pelo compilador.
input	30 input x	Exibe um ponto de interrogação para pedir que o usuário insira um inteiro. Lê esse inteiro a partir do teclado e o armazena em x.
let	80 let u = 4 * (j - 56)	Atribui a u o valor de $4 * (j - 56)$ . Observe que uma expressão arbitrariamente complexa pode aparecer à direita do sinal de igual.
print	10 print w	Exibe o valor de w.
goto	70 goto 45	Transfere o controle do programa para a linha 45.
if...goto	35 if i == z goto 80	Compara i e z quanto à igualdade e transfere o controle para a linha 80 se a condição for verdadeira; caso contrário, continua a execução com a próxima instrução.
end	99 end	Termina execução do programa.

**Figura 21.25** Comandos do Simple.

como zero automaticamente. A sintaxe do Simple não permite manipulação de string (ler uma string, gravar uma string, comparar strings etc.) Se uma string for encontrada em um programa Simple (após qualquer outro comando que não rem), o compilador gera um erro de sintaxe. A primeira versão de nosso compilador supõe que os programas Simple são digitados corretamente. O Exercício 21.29 pede para o aluno modificar o compilador para realizar verificação de erros de sintaxe.

O Simple utiliza a instrução condicional if...goto e a instrução incondicional goto para alterar o fluxo de controle durante a execução do programa. Se a condição na instrução if...goto for verdadeira, o controle é transferido para uma linha específica do programa. Os seguintes operadores relacionais de igualdade são válidos em uma instrução if...goto: <, >, <=, >=, == e !=. A precedência desses operadores é a mesma que em C++.

Vamos agora considerar vários programas que demonstram recursos do Simple. O primeiro programa (Figura 21.26) lê dois inteiros do teclado, armazena os valores nas variáveis a e b e calcula e imprime sua soma (armazenada na variável c).

O programa da Figura 21.27 determina e imprime o maior de dois inteiros. Os inteiros são inseridos do teclado e armazenados em s e t. A instrução if...goto testa a condição  $s \geq t$ . Se a condição for verdadeira, o controle é transferido para a linha 90, e s é enviado para a saída; caso contrário, t é enviado para a saída, e o controle é transferido para a instrução end na linha 99, onde o programa termina.

O Simple não fornece uma instrução de repetição (como for, while ou do...while do C++). Entretanto, o Simple pode simular cada uma das instruções de repetição do C++ utilizando as instruções if...goto e goto. A Figura 21.28 utiliza um loop controlado por sentinela para calcular os quadrados de vários inteiros. Cada inteiro é inserido do teclado e armazenado na variável j. Se o valor inserido é o valor de sentinela -9999, o controle é transferido para a linha 99, onde o programa termina. Caso contrário, k é atribuído ao quadrado de j, k é enviado para a saída de tela e o controle é passado para a linha 20, onde o próximo inteiro é inserido.

Utilizando os programas de exemplo das figuras 21.26, 21.27 e 21.28 como seu guia, escreva um programa Simple para realizar cada uma das seguintes instruções:

- Insira três inteiros, determine sua média e imprima o resultado.
- Utilize um loop controlado por sentinela para inserir 10 inteiros e calcular e imprimir sua soma.
- Utilize um loop controlado por contador para inserir sete inteiros, alguns positivos e alguns negativos, e calcule e imprima sua média.
- Insira uma série de inteiros e determine e imprima o maior. A primeira entrada de inteiro indica quantos números devem ser processados.
- Insira 10 inteiros e imprima o menor.
- Calcule e imprima a soma dos inteiros pares de 2 a 30.
- Calcule e imprima o produto dos inteiros ímpares de 1 a 9.

**21.27** (Construindo um compilador; Pré-requisito: Completar os exercícios 8.18, 8.19, 21.12, 21.13 e 21.26) Agora que a linguagem Simple foi apresentada (Exercício 21.26), discutiremos como construir um compilador de Simple. Primeiro, consideramos o processo pelo qual um programa Simple é convertido em SML e executado pelo Simpletron Simulator (ver Figura 21.29). Um arquivo contendo um programa Simple é lido pelo compilador e convertido em código SML. O código SML é enviado para um arquivo de saída em disco, no qual instruções de SML aparecem uma em cada linha. O arquivo SML então é carregado no Simpletron Simulator e os resultados são enviados para

```

1 10 rem determine and print the sum of two integers
2 15 rem
3 20 rem input the two integers
4 30 input a
5 40 input b
6 45 rem
7 50 rem add integers and store result in c
8 60 let c = a + b
9 65 rem
10 70 rem print the result
11 80 print c
12 90 rem terminate program execution
13 99 end

```

**Figura 21.26** Programa em Simple que determina a soma de dois inteiros.

```

1 10 rem determine the larger of two integers
2 20 input s
3 30 input t
4 32 rem
5 35 rem test if s >= t
6 40 if s >= t goto 90
7 45 rem
8 50 rem t is greater than s, so print t
9 60 print t
10 70 goto 99
11 75 rem
12 80 rem s is greater than or equal to t, so print s
13 90 print s
14 99 end

```

**Figura 21.27** Programa em Simple que localiza o maior de dois inteiros.

```

1 10 rem calculate the squares of several integers
2 20 input j
3 23 rem
4 25 rem test for sentinel value
5 30 if j == -9999 goto 99
6 33 rem
7 35 rem calculate square of j and assign result to k
8 40 let k = j * j
9 50 print k
10 53 rem
11 55 rem loop to get next j
12 60 goto 20
13 99 end

```

**Figura 21.28** Calcule os quadrados de vários inteiros.

um arquivo em disco e para a tela. Observe que o programa Simpletron desenvolvido no Exercício 8.19 pegou sua entrada do teclado. Ele deve ser modificado para ler de um arquivo de modo que possa executar os programas produzidos pelo nosso compilador.

O compilador de Simple realiza duas *passagens* do programa Simple para convertê-lo em SML. A primeira passagem constrói uma *tabela de símbolos* (objeto) em que cada *número de linha* (objeto), *nome de variável* (objeto) e *constante* (objeto) do programa Simple é armazenado com seu tipo e posição correspondente no código SML final (a tabela de símbolos é discutida em detalhe a seguir). A primeira passagem também produz o(s) correspondente(s) objeto(s) de instrução (*instruction*) de SML para cada uma das instruções (*statements*) do Simple (objeto etc.). Como veremos, se o programa Simple contém instruções que transferem o controle posteriormente para uma linha no programa, a primeira passagem resulta em um programa SML contendo parte das instruções ‘não concluídas’. A segunda passagem do compilador localiza e completa as instruções não finalizadas e envia o programa SML para um arquivo de saída.

### Primeira passagem

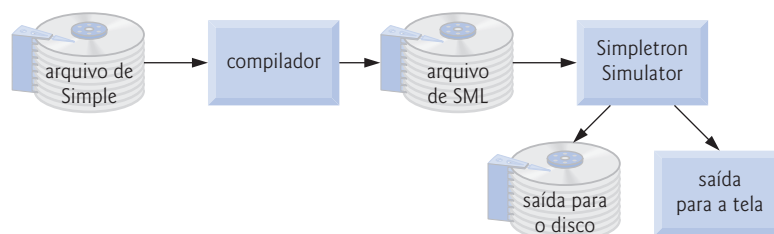
O compilador começa a ler uma instrução (*statement*) do programa Simple na memória. A linha deve ser separada em seus *tokens* individuais (isto é, ‘fragmentos’ de uma instrução) para processamento e compilação (a função `strtok` da biblioteca-padrão pode ser utilizada para facilitar essa tarefa). Lembre-se de que cada instrução (*statement*) inicia com um número de linha seguido por um comando. Quando o compilador divide uma instrução (*statement*) em tokens, se o token é um número de linha, uma variável ou uma constante, ele é colocado na tabela de símbolos. Um número de linha é colocado na tabela de símbolos apenas se for o primeiro token em uma instrução. O objeto `symbolTable` é um array de objetos `tableEntry` representando cada símbolo no programa. Não há restrição quanto ao número de símbolos que podem aparecer no programa. Portanto, a `symbolTable` para um programa particular pode ser grande. Torne a `symbolTable` um array de 100 elementos por enquanto. Você pode aumentar ou diminuir seu tamanho depois que o programa estiver funcionando.

Cada objeto `tableEntry` contém três membros. O membro `symbol` é um inteiro contendo a representação ASCII de uma variável (lembre-se de que os nomes de variável são caracteres únicos), um número de linha ou uma constante. O membro `type` é um dos seguintes caracteres que indicam o tipo do símbolo: 'C' para constante, 'L' número de linha e 'V' para variável. O membro `location` contém a posição da memória do Simpletron (00 para 99) que o símbolo referencia. A memória do Simpletron é um array de 100 inteiros em que as instruções de SML e os dados são armazenados. Para um número de linha, a posição é o elemento no array de memória do Simpletron em que as instruções de SML para a instrução Simple iniciam. Para uma variável ou constante, a posição é o elemento no array de memória de Simpletron em que a variável ou constante é armazenada. Variáveis e constantes são alocadas a partir do fim da memória do Simpletron para trás. A primeira variável ou constante é armazenada na posição em 99, a próxima na posição em 98 etc.

A tabela de símbolos desempenha uma parte integrante na conversão de programas Simple em SML. Aprendemos no Capítulo 8 que uma instrução SML é um inteiro de quatro dígitos compostos de duas partes — o *código de operação* e o *operando*. O código de operação é determinado por comandos em Simple. Por exemplo, o comando `input` do Simple corresponde ao código de operação 10 (*read*) do SML e o comando `print` do Simple corresponde ao código de operação 11 (*write*) de SML. O operando é uma posição da memória contendo os dados em que o código de operação realiza sua tarefa (por exemplo, o código de operação 10 lê um valor do teclado e o armazena na posição da memória especificada pelo operando). O compilador pesquisa `symbolTable` para determinar a posição da memória no Simpletron para cada símbolo, de modo que a posição correspondente possa ser utilizada para completar as instruções de SML.

A compilação de cada instrução do Simple é baseada em seu comando. Por exemplo, depois que o número de linha em uma instrução `rem` é inserido na tabela de símbolos, o restante da instrução é ignorado pelo compilador porque uma observação serve apenas para propósitos de documentação. As instruções `input`, `print`, `goto` e `end` correspondem às instruções do SML *read*, *write*, *branch* ([desviar] para uma posição específica) e *halt*. As instruções contendo esses comandos do Simple são convertidas diretamente em SML (note que uma instrução `goto` pode conter uma referência não resolvida se o número de linha especificado referencia uma instrução mais adiante no arquivo do programa Simple; isso é às vezes chamado de referência antecipada).

Quando uma instrução `goto` é compilada com uma referência não resolvida, a instrução de SML deve ser *marcada com um flag* para indicar que a segunda passagem do compilador deve completar a instrução. Os flags são armazenados em `flags` de array de 100 elemen-



**Figura 21.29** Gravando, compilando e executando um programa da linguagem Simple.

tos do tipo `int` em que cada elemento é inicializado como `-1`. Se a posição da memória a que um número de linha no programa Simple se refere ainda não é conhecida (isto é, não está na tabela de símbolos), o número de linha é armazenado em `flags` array no elemento com o mesmo subscrito que a instrução incompleta. O operando da instrução incompleta é configurado como `00` temporariamente. Por exemplo, uma instrução de desvio incondicional (*unconditional branch*, fazer uma referência antecipada) é deixada como `+4000` até a segunda passagem do compilador. A segunda passagem do compilador será descrita em breve.

A compilação das instruções `if...goto` e `let` é mais complicada que outras instruções — elas são as únicas instruções que produzem mais de uma instrução SML. Para um `if...goto`, o compilador produz código para testar a condição e desviar para outra linha se necessário. O resultado do desvio pode ser uma referência não resolvida. Cada um dos operadores relacionais de igualdade pode ser simulado utilizando instruções de *branch zero* ou *branch negative* do SML (ou uma combinação das duas).

Para uma instrução `let`, o compilador produz código para avaliar uma expressão aritmética arbitrariamente complexa consistindo em variáveis e/ou constantes de inteiro. As expressões devem separar cada operando e operador com espaços. Os exercícios 21.12 e 21.13 apresentaram o algoritmo de conversão de infixo em pós-fixado e o algoritmo de avaliação pós-fixado utilizado por compiladores para avaliar expressões. Antes de prosseguir com seu compilador, você deve completar cada um desses exercícios. Quando um compilador encontra uma expressão, ele converte a expressão de notação infixa em notação pós-fixada, e então avalia a expressão pós-fixada.

Como o compilador produz a linguagem de máquina para avaliar uma expressão contendo variáveis? O algoritmo de avaliação de pós-fixado contém um ‘gancho’ onde o compilador pode gerar instruções de SML em vez de realmente avaliar a expressão. Para ativar esse ‘gancho’ no compilador, o algoritmo de avaliação pós-fixado deve ser modificado para pesquisar a tabela de símbolos para cada símbolo que ele encontra (e possivelmente inseri-lo), determinar a posição da memória correspondente do símbolo e *adicionar a posição da memória na pilha (em vez do símbolo)*. Quando um operador é encontrado na expressão pós-fixada, as duas posições da memória na parte superior da pilha são removidas e a linguagem de máquina para afetar a operação é produzida, utilizando as posições da memória como operandos. O resultado de cada subexpressão é armazenado em uma posição temporária na memória e inserido de volta na pilha de tal modo que a avaliação da expressão pós-fixada possa continuar. Quando a avaliação pós-fixada está completa, a posição da memória contendo o resultado é a única posição deixada na pilha. Essa é removida e as instruções de SML são geradas para atribuir o resultado à variável à esquerda da instrução `let`.

### Segunda passagem

A segunda passagem do compilador realiza duas tarefas: resolver quaisquer referências não resolvidas e enviar o código de SML para um arquivo de saída. A solução de referências ocorre como segue:

- Pesquise no array `flags` uma referência não resolvida (isto é, um elemento com um valor outro que `-1`).
- Localize o objeto no array `symbolTable` contendo o símbolo armazenado no array `flags` (certificando-se de que o tipo do símbolo é ‘L’ para o número de linha).
- Insira a posição de memória do membro `location` na instrução com a referência não resolvida (lembre-se de que uma instrução contendo uma referência não resolvida tem operando `00`).
- Repita os Passos 1, 2 e 3 até o final do array `flags` ser alcançado.

Depois que o processo de solução estiver completo, o array inteiro contendo o código de SML é enviado para um arquivo de saída em disco com uma instrução de SML por linha. Esse arquivo pode ser lido pelo Simpletron para execução (depois que o simulador é modificado para ler sua entrada de um arquivo). Compilar seu primeiro programa Simple em um arquivo de SML e então executar esse arquivo deve lhe dar uma verdadeira sensação de realização pessoal.

### Um exemplo completo

O seguinte exemplo ilustra a conversão completa de um programa Simple em SML como ele será realizado pelo compilador de Simple. Considere um programa Simple que insere um inteiro e soma os valores de 1 até esse inteiro. O programa e as instruções de SML produzidas pela primeira passagem do compilador de Simple são ilustrados na Figura 21.30. A tabela de símbolos construída pela primeira passagem é mostrada na Figura 21.31.

A maioria das instruções do Simple é convertida diretamente em instruções únicas de SML. As exceções nesse programa são os comentários, a instrução `if...goto` na linha 20 e as instruções `let`. Os comentários não são traduzidos para a linguagem de máquina. Entretanto, o número de linha para uma observação é colocado na tabela de símbolos no caso de o número de linha ser referenciado em uma instrução `goto` ou em uma instrução `if...goto`. A linha 20 do programa especifica que, se a condição `y == x` for verdadeira, o controle do programa é transferido para a linha 60. Como a linha 60 aparece mais adiante no programa, a primeira passagem do compilador ainda não colocou 60 na tabela de símbolos (os números de linha de instrução são colocados na tabela de símbolos apenas quando aparecem como o primeiro token em uma instrução). Portanto, não é possível nesse momento determinar o operando da instrução *branch zero* do SML na posição 03 no array de instruções de SML. O compilador coloca 60 na posição 03 do array `flags` para indicar que a segunda passagem completa essa instrução.

Devemos monitorar a próxima posição da instrução no array SML porque não há uma correspondência um para um entre as instruções de Simple e as instruções de SML. Por exemplo, a instrução `if...goto` da linha 20 é compilada em três instruções de SML. Toda vez que uma instrução é produzida, devemos incrementar o *contador de instrução* para a próxima posição no array de SML. Observe que o

Programa Simple	Posição & instrução SML	Descrição
5 rem sum 1 to x	<i>nenhuma</i>	rem ignorado
10 input x	00 +1099	lê x na posição 99
15 rem check y == x	<i>nenhuma</i>	rem ignorado
20 if y == x goto 60	01 +2098	carrega y (98) no acumulador
	02 +3199	subtrai x (99) a partir do acumulador
	03 +4200	desvia se zero para posição não resolvida
25 rem increment y	<i>nenhuma</i>	rem ignorado
30 let y = y + 1	04 +2098	carrega y no acumulador
	05 +3097	adiciona 1 (97) ao acumulador
	06 +2196	armazena na localização temporária 96
	07 +2096	carrega a partir da posição temporária 96
	08 +2198	armazena acumulador em y
35 rem add y to total	<i>nenhuma</i>	rem ignorado
40 let t = t + y	09 +2095	carrega t (95) no acumulador
	10 +3098	adiciona y ao acumulador
	11 +2194	armazena na localização temporária 94
	12 +2094	carrega a partir da posição temporária 94
	13 +2195	armazena acumulador em t
45 rem loop y	<i>nenhuma</i>	rem ignorado
50 goto 20	14 +4001	desvia para a posição 01
55 rem output result	<i>nenhuma</i>	rem ignorado
60 print t	15 +1195	saída t para a tela
99 end	16 +4300	termina a execução

**Figura 21.30** Instruções de SML produzidas depois da primeira passagem do compilador.

Símbolo	Tipo	Posição	Símbolo	Tipo	Posição
5	L	00	35	L	09
10	L	00	40	L	09
'x'	V	99	't'	V	95
15	L	01	45	L	14
20	L	01	50	L	14
'y'	V	98	55	L	15
25	L	04	60	L	15
30	L	04	99	L	16
1	C	97			

**Figura 21.31** A tabela de símbolos para programa da Figura 21.30.

tamanho da memória do Simpletron pode apresentar um problema para programas Simple com muitas instruções, variáveis e constantes. É concebível que o compilador fique sem memória. Para testar esse caso, seu programa deve conter um *contador de dados* para monitorar a posição em que a próxima variável ou constante será armazenada no array de SML. Se o valor do contador de instrução for maior que o valor do contador de dados, o array de SML está cheio. Nesse caso, o processo de compilação deve terminar e o compilador deve imprimir uma mensagem de erro indicando que ele ficou sem memória durante a compilação. Isso serve para enfatizar que, embora o programador seja liberado do peso de gerenciar a memória pelo compilador, o próprio compilador deve cuidadosamente determinar a colocação de instruções e dados na memória e verificar tais erros quando a memória se esgota durante o processo de compilação.

### *Uma visualização passo a passo do processo de compilação*

Vamos agora percorrer o processo de compilação para o programa Simple na Figura 21.30. O compilador lê a primeira linha do programa

```
5 rem sum 1 to x
```

para a memória. O primeiro token na instrução (o número da linha) é determinado utilizando `strtok` (ver os capítulos 8 e 21 para uma discussão sobre as funções de manipulação de strings no estilo C do C++). O token retornado por `strtok` é convertido em um inteiro que utiliza `atoi`, portanto o símbolo 5 pode ser localizado na tabela de símbolos. Se o símbolo não for localizado, ele é inserido na tabela de símbolos. Já que estamos no começo do programa e essa é a primeira linha, nenhum símbolo ainda está na tabela. Então, 5 é inserido na tabela de símbolos como o tipo L (número da linha) e atribuído à primeira posição no array de SML (00). Embora essa linha seja uma observação, um espaço na tabela de símbolos ainda é alocado para o número de linha (no caso, ele é referenciado por um `goto` ou um `if...goto`). Nenhuma instrução de SML é gerada para uma instrução `rem`, então o contador de instrução não é incrementado.

A instrução

```
10 input x
```

é ‘tokenizada’ (separada em tokens, ou marcas) a seguir. O número de linha 10 é colocado na tabela de símbolos como o tipo L e atribuído à primeira posição no array de SML (00, porque um comentário foi iniciado no programa, então o contador de instrução é atualmente 00). O comando `input` indica que o próximo token é uma variável (apenas uma variável pode aparecer em uma instrução `input`). Como `input` corresponde diretamente a um código de operação de SML, o compilador tem de determinar a posição de `x` no array de SML. O símbolo `x` não foi encontrado na tabela de símbolos, portanto ele é inserido na tabela de símbolos como a representação ASCII de `x`, recebe o tipo V e é atribuído à posição 99 no array SML (o armazenamento de dados começa em 99 e é alocado de trás para a frente). O código de SML agora pode ser gerado para essa instrução. O código de operação 10 (o código de operação de leitura do SML) é multiplicado por 100 e a posição de `x` (como determinado na tabela de símbolos) é adicionada para completar a instrução. A instrução então é armazenada no array de SML na posição 00. O contador de instrução é incrementado por 1, porque uma única instrução de SML foi produzida.

A instrução

```
15 rem check y == x
```

é ‘tokenizada’ (separada em tokens, ou marcas) a seguir. A tabela de símbolos é pesquisada por número de linha 15 (que não é localizado). O número da linha é inserido como o tipo L e recebe a próxima posição no array, 01 (lembre-se de que as instruções `rem` não produzem código, então o contador de instruções não é incrementado).

A instrução

```
20 if y == x goto 60
```

é ‘tokenizada’ (separada em tokens, ou marcas) a seguir. O número da linha 20 é inserido na tabela de símbolos e o tipo dado L com a próxima posição no array de SML 01. O comando `if` indica que uma condição será avaliada. A variável `y` não é localizada na tabela de símbolos, então é inserida e `lhe` são atribuídos o tipo V e a posição de SML 98. Em seguida, instruções de SML são geradas para avaliar a condição. Como não há equivalente direto em SML a `if...goto`, ele deve ser simulado realizando um cálculo com `x` e `y` e desviando com base no resultado. Se `y` for igual a `x`, o resultado de subtrair `x` de `y` é zero, então a instrução *branch zero* pode ser utilizada com o resultado do cálculo para simular a instrução `if...goto`. O primeiro passo requer que `y` seja carregado (da posição 98 do SML) no acumulador. Isso produz a instrução 01 +2098. Em seguida, `x` é subtraído do acumulador. Isso produz a instrução 02 +3199. O valor no acumulador pode ser zero, positivo ou negativo. Como o operador é `==`, queremos *branch zero*. Primeiro, a tabela de símbolos é pesquisada quanto à posição do desvio (60 nesse caso), que não é localizada. Assim, 60 é colocado no array `flags` na posição 03, e a instrução 03 +4200 é gerada (não podemos adicionar a posição de desvio porque ainda não atribuímos uma posição à linha 60 no array de SML). O contador de instrução é incrementado para 04.

O compilador prossegue para instrução

```
25 rem increment y
```

O número de linha 25 é inserido na tabela de símbolos como o tipo L e atribuído à posição 04 da SML. O contador de instrução não é incrementado.

Quando a instrução



```
30 let y = y + 1
```

é ‘tokenizada’ (dividida em tokens), o número de linha 30 é inserido na tabela de símbolos como o tipo L e é atribuído à posição 04 da SML. O comando `let` indica que a linha é uma instrução de atribuição. Primeiro, todos os símbolos na linha são inseridos na tabela de símbolos (se ainda não estiverem aí). O inteiro 1 é adicionado à tabela de símbolos como o tipo C e é atribuído à posição 97 da SML. Em seguida, o lado direito da atribuição é convertido de notação infixa em pós-fixa. Então a expressão pós-fixa (`y 1 +`) é avaliada. O símbolo `y` é localizado na tabela de símbolos e sua posição da memória correspondente é inserida na pilha. O símbolo 1 também é localizado na tabela de símbolos e sua correspondente posição na memória é inserida na pilha. Quando o operador `+` é encontrado, o avaliador de pós-fixa remove da pilha o operando direito do operador, remove da pilha novamente o operando esquerdo do operador e produz as instruções de SML

```
04 +2098 (load y)
05 +3097 (add 1)
```

O resultado da expressão é armazenado em uma posição temporária na memória (96) com a instrução

```
06 +2196 (store temporary)
```

e a posição temporária é adicionada à pilha. Agora que a expressão foi avaliada, o resultado deve ser armazenado em `y` (isto é, a variável no lado esquerdo de `=`). Então, a posição temporária é carregada no acumulador e o acumulador é armazenado em `y` com as instruções

```
07 +2096 (load temporary)
08 +2198 (store y)
```

O leitor imediatamente notará que as instruções de SML parecem ser redundantes. Discutiremos essa questão brevemente.

Quando a instrução

```
35 rem add y to total
```

é tokenizada (dividida em tokens), o número de linha 35 é inserido na tabela de símbolos como o tipo L e atribuído à posição 09.

A instrução

```
40 let t = t + y
```

é semelhante à linha 30. A variável `t` é inserida na tabela de símbolos como o tipo V e atribuída à posição 95 da SML. As instruções seguem a mesma lógica e formato que a linha 30 e as instruções 09 +2095, 10 +3098, 11 +2194, 12 +2094 e 13 +2195 são geradas. Observe que o resultado de `t + y` é atribuído à posição temporária 94 antes de ser atribuído a `t` (95). Mais uma vez, o leitor notará que as instruções nas posições da memória 11 e 12 parecem ser redundantes. Novamente, discutiremos isso em breve.

A instrução

```
45 rem loop y
```

é um comentário, então a linha 45 é adicionada à tabela de símbolos como o tipo L e atribuída à posição 14 da SML.

A instrução

```
50 goto 20
```

transfere o controle para a linha 20. O número de linha 50 é armazenado na tabela de símbolos como o tipo L e atribuído à posição 14 da SML. O equivalente de `goto` em SML é a instrução *unconditional branch* (40), que transfere o controle para uma posição específica de SML. O compilador pesquisa a tabela de símbolos para a linha 20 e acha que corresponde à posição 01 de SML. O código de operação (40) é multiplicado por 100 e a posição 01 é adicionada para produzir a instrução 14 +4001.

A instrução

```
55 rem output result
```

é um comentário, então a linha 55 é inserida na tabela de símbolos como o tipo L e é atribuída à posição 15 da SML.

A instrução

```
60 print t
```

é uma instrução de saída. O número de linha 60 é armazenado na tabela de símbolos como o tipo L e atribuído à posição 15 da SML. O equivalente de `print` na SML é o código de operação 11 (*write*). A posição de `t` é determinada a partir da tabela de símbolos e adicionada ao resultado do código de operação multiplicado por 100.

A instrução

```
99 end
```

é a linha final do programa. O número de linha 99 é armazenado na tabela de símbolos como o tipo L e atribuído à posição 16 de SML. O comando `end` produz a instrução de SML +4300 (43 é *halt* na SML), que é escrita como a instrução final no array de memória de SML.

Isso completa a primeira passagem do compilador. Agora consideramos a segunda passagem. O array `flags` é pesquisado quanto a outros valores que não `-1`. A posição 03 contém 60, então o compilador sabe que a instrução 03 está incompleta. O compilador completa a instrução pesquisando 60 na tabela de símbolos, determinando sua posição e adicionando a posição à instrução incompleta. Nesse caso, a pesquisa determina que a linha 60 corresponde à posição 15 de SML, então a instrução completada 03 +4215 é produzida, substituindo 03 +4200. O programa Simple agora compilou com sucesso.

Para construir o compilador, você terá de realizar cada uma das seguintes tarefas:

- Modifique o programa do Simpletron Simulator que você escreveu no Exercício 8.19 para pegar sua entrada de um arquivo especificado pelo usuário (veja o Capítulo 17). O simulador deve enviar seus resultados para um arquivo de saída em disco no mesmo formato que a saída em tela. Converta o simulador para ser um programa orientado a objetos. Em particular, torne cada parte do hardware um objeto. Organize os tipos de instrução em uma hierarquia de classes utilizando herança. Então execute o programa polimorficamente fazendo cada instrução executar a si própria com uma mensagem `executeInstruction`.
- Modifique o algoritmo de conversão de infix para pós-fixado do Exercício 21.12 para processar operandos de inteiro de múltiplos dígitos e operandos de nome de variável de uma única letra. [Dica: A função `strtok` da C++ Standard Library pode ser utilizada para localizar cada constante e variável em uma expressão, e as constantes podem ser convertidas de strings para inteiros utilizando a função da biblioteca-padrão `atoi` (`<csdtlib>`).] [Nota: A representação de dados da expressão pós-fixada deve ser alterada para suportar nomes de variáveis e constantes de inteiro.]
- Modifique o algoritmo de avaliação pós-fixada para processar operandos de inteiro de múltiplos dígitos e operandos de nome de variável. Além disso, o algoritmo agora deve implementar o 'gancho' discutido anteriormente de modo que sejam produzidas instruções de SML em vez de diretamente avaliar a expressão. [Dica: A função da biblioteca-padrão `strtok` pode ser utilizada para localizar cada constante e variável em uma expressão, e as constantes podem ser convertidas de strings para inteiros utilizando a função da biblioteca-padrão `atoi`.] [Nota: A representação de dados da expressão pós-fixada deve ser alterada para suportar nomes de variáveis e constantes de inteiro.]
- Construa o compilador. Incorpore as partes (b) e (c) para avaliar as expressões em instruções `let`. Seu programa deve conter uma função que realiza a primeira passagem do compilador e uma função que realiza a segunda passagem do compilador. Ambas as funções podem chamar outras funções para realizar suas tarefas. Torne seu compilador orientado a objeto o máximo possível.

**21.28** (*Otimizando o compilador de Simple*) Quando um programa é compilado e convertido em SML, um conjunto de instruções é gerado. Certas combinações de instruções freqüentemente se repetem, normalmente em triplos chamados *produções*. Uma produção normalmente consiste em três instruções, como *load*, *add* e *store*. Por exemplo, a Figura 21.32 ilustra cinco das instruções da SML que foram produzidas na compilação do programa na Figura 21.30. As primeiras três instruções são a produção que adiciona 1 a *y*. Observe que as instruções 06 e 07 armazenam o valor do acumulador na posição temporária 96 e carregam o valor de volta no acumulador de modo que a instrução 08 possa armazenar o valor na posição 98. Frequentemente uma produção é seguida por uma instrução de carregar na mesma posição que acabou de ser armazenada. Esse código pode ser *otimizado* eliminando a instrução de armazenar e a instrução subsequente de carregar que opera na mesma posição da memória, permitindo assim a Simpletron executar o programa mais rápido. A Figura 21.33 ilustra a SML otimizada para o programa da Figura 21.30. Observe que há menos quatro instruções no código otimizado — uma economia de 25% de espaço de memória.

Modifique o compilador para fornecer uma opção para otimizar o código que a Simpletron Machine Language produz. Compare manualmente o código não otimizado com o código otimizado e calcule a redução de porcentagem.

**21.29** (*Modificações no compilador de Simple*) Realize as seguintes modificações no compilador de Simple. Algumas dessas modificações também podem exigir modificações no programa Simpletron Simulator escrito no Exercício 8.19.

- Permita que o operador de módulo (%) seja utilizado em instruções `let`. A Simpletron Machine Language deve ser modificada para incluir uma instrução de módulo.
- Permita exponenciação em uma instrução `let` utilizando ^ como o operador de exponenciação. A Simpletron Machine Language deve ser modificada para incluir uma instrução de exponenciação.
- Permita que o compilador reconheça letras minúsculas e maiúsculas em instruções Simple (por exemplo, 'A' é equivalente a 'a'). Nenhuma modificação no Simulator é necessária.
- Permita que as instruções `input` leiam os valores e os transfiram para múltiplas variáveis como `input x, y`. Nenhuma modificação no Simpletron Simulator é necessária.

```

1 04 +2098 (load)
2 05 +3097 (add)
3 06 +2196 (store)
4 07 +2096 (load)
5 08 +2198 (store)
```

**Figura 21.32** Código não otimizado do programa da Figura 21.30.

Programa Simple	Posição & instrução SML	Descrição
5 rem sum 1 to x	<i>nenhuma</i>	rem ignorado
10 input x	00 +1099	lê x na posição 99
15 rem check y == x	<i>nenhuma</i>	rem ignorado
20 if y == x goto 60	01 +2098	carrega y (98) no acumulador
	02 +3199	subtrai x (99) do acumulador
	03 +4211	desvia para a posição 11 se zero
25 rem increment y	<i>nenhuma</i>	rem ignorado
30 let y = y + 1	04 +2098	carrega y no acumulador
	05 +3097	adiciona 1 (97) ao acumulador
	06 +2198	armazena o acumulador em y (98)
35 rem add y to total	<i>nenhuma</i>	rem ignorado
40 let t = t + y	07 +2096	carrega t a partir da posição (96)
	08 +3098	adiciona y (98) ao acumulador
	09 +2196	armazena o acumulador em t (96)
45 rem loop y	<i>nenhuma</i>	rem ignorado
50 goto 20	10 +4001	desvia para a posição 01
55 rem output result	<i>nenhuma</i>	rem ignorado
60 print t	11 +1196	gera saída de t (96) para tela
99 end	12 +4300	termina a execução

**Figura 21.33** Código otimizado para o programa da Figura 21.30.

- e) Permita que o compilador gere saída de múltiplos valores em uma única instrução `print`, como `print a, b, c`. Nenhuma modificação no Simpletron Simulator é necessária.
- f) Adicione capacidades de verificação de sintaxe ao compilador de modo que as mensagens de erro sejam enviadas para a saída quando erros de sintaxe forem encontrados em um programa Simple. Nenhuma modificação no Simpletron Simulator é necessária.
- g) Permita arrays de inteiros. Nenhuma modificação no Simpletron Simulator é necessária.
- h) Permita sub-rotinas especificadas pelos comandos `gosub` e `return` do Simple. O comando `gosub` passa o controle do programa para uma sub-rotina e o comando `return` passa o controle de volta à instrução depois do `gosub`. Isso é semelhante a uma chamada de função em C++. A mesma sub-rotina pode ser chamada de muitos comandos `gosub` distribuídos por todo um programa. Nenhuma modificação no Simpletron Simulator é necessária.
- i) Permita instruções de repetição da fórmula

```

for x = 2 to 10 step 2
 Simple statements
next

```

Essa instrução `for` faz loop de 2 a 10 com um incremento de 2. A linha `next` marca o fim do corpo do `for`. Nenhuma modificação no Simpletron Simulator é necessária.

- j) Permita instruções de repetição da fórmula

```

for x = 2 to 10
 Simple statements
next

```

Essa instrução `for` faz loop de 2 a 10 com um incremento-padrão de 1. Nenhuma modificação no Simpletron Simulator é necessária.

- k) Permita que o compilador processe entrada e saída de string. Isso requer que o Simpletron Simulator seja modificado para processar e armazenar valores de string. [Dica: Cada palavra do Simpletron pode ser dividida em dois grupos, cada um armazenando um inteiro de dois dígitos. Cada inteiro de dois dígitos representa o equivalente ASCII decimal de um caractere. Adicione uma instrução de linguagem de máquina que imprimirá uma string inicial em certa posição da memória de Simpletron. A primeira metade da palavra nessa posição é uma contagem do número de caracteres na string (isto é, o comprimento da string). Cada meia palavra sucessiva contém um caractere ASCII expresso como dois dígitos decimais. A instrução de linguagem de máquina verifica o comprimento e imprime a string traduzindo cada número de dois dígitos em seu caractere equivalente.]
- l) Permita que o compilador processe valores de ponto flutuante além de inteiros. O Simpletron Simulator também deve ser modificado para processar valores de ponto flutuante.

**21.30** (*Um interpretador de Simple*) Um interpretador é um programa que lê uma instrução de programa de uma linguagem de alto nível, determina a operação a ser realizada pela instrução e executa a operação imediatamente. O programa de linguagem de alto nível não é convertido em linguagem de máquina primeiro. Os interpretadores executam lentamente porque cada instrução encontrada no programa deve ser decifrada primeiro. Se as instruções estão contidas em um loop, as instruções são decifradas toda vez que são encontradas no loop. As versões anteriores da linguagem de programação BASIC foram implementadas como interpretadores.

Escreva um interpretador para a linguagem Simple discutida no Exercício 21.26. O programa deve utilizar o conversor de infixo para pós-fixado desenvolvido no Exercício 21.12 e o avaliador de pós-fixado desenvolvido no Exercício 21.13 para avaliar expressões em uma instrução `let`. As mesmas restrições impostas à linguagem Simple do Exercício 21.26 devem ser obedecidas nesse programa. Teste o interpretador com os programas Simple escritos no Exercício 21.26. Compare os resultados de executar esses programas no interpretador com os resultados de compilar os programas Simple e executá-los no Simpletron Simulator construído no Exercício 8.19.

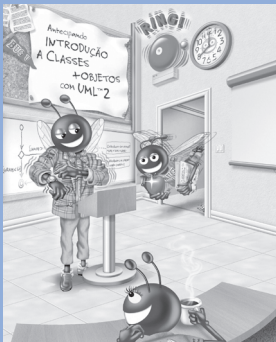
**21.31** (*Inserção/exclusão em qualquer lugar em uma lista vinculada*) Nosso template de classe de listas vinculadas permitia inserções e exclusões no início e no fim da lista vinculada. Essas capacidades foram convenientes quando utilizamos a herança `private` e a composição para produzir um template de classe de pilhas e um template de classe de filas com uma quantidade mínima de código reutilizando o template de classe de lista. Na realidade, as listas vinculadas são mais gerais que aquelas que fornecemos. Modifique o template de classe de listas vinculadas que desenvolvemos neste capítulo para tratar inserções e exclusões em qualquer lugar na lista.

**21.32** (*Lista e filas sem ponteiros de cauda*) Nossa implementação de uma lista vinculada (figuras 21.3–21.5) utilizou tanto um `firstPtr` como um `lastPtr`. O `lastPtr` foi útil para as funções-membro `insertAtBack` e `removeFromBack` da classe `List`. A função `insertAtBack` corresponde à função-membro `enqueue` da classe `Queue`. Reescreva a classe `List` de modo que ela não utilize um `lastPtr`. Portanto, quaisquer operações no fim de uma lista devem começar pesquisando no início da lista. Isso afeta nossa implementação da classe `Queue` (Figura 21.16)?

**21.33** Utilize a versão composta do programa de pilha (Figura 21.15) para formar um programa de pilha funcional completo. Modifique esse programa para funções-membro `inline`. Compare as duas abordagens. Resuma as vantagens e desvantagens de colocar `inline` funções-membro.

**21.34** (*Desempenho da classificação e da pesquisa de árvore binária*) Um problema com a classificação de árvore binária é que a ordem em que os dados são inseridos afeta a forma da árvore — para a mesma coleção de dados, diferentes ordens podem produzir árvores binárias de formas significativamente diferentes. O desempenho dos algoritmos de classificação e pesquisa de árvore binária é sensível à forma da árvore binária. Que forma teria uma árvore binária se seus dados fossem inseridos na ordem crescente? E na ordem decrescente? Que forma a árvore deveria ter para alcançar desempenho máximo de pesquisa?

**21.35** (*Listas indexadas*) Como apresentado no texto, as listas vinculadas devem ser pesquisadas sequencialmente. Para listas grandes, isso pode resultar em desempenho pobre. Uma técnica comum para aprimorar o desempenho de pesquisa de lista é criar e manter um índice para a lista. Um índice é um conjunto de ponteiros para vários lugares-chave na lista. Por exemplo, um aplicativo que pesquisa uma grande lista de nomes pode aprimorar seu desempenho criando um índice com 26 entradas — uma para cada letra do alfabeto. Uma operação de pesquisa de um sobrenome que inicia com ‘Y’ primeiro pesquisaria o índice para determinar onde as entradas ‘Y’ iniciam e então ‘saltaria’ na lista nesse ponto e pesquisaria linearmente até que o nome desejado fosse localizado. Isso seria muito mais rápido que pesquisar a lista vinculada desde o início. Utilize a classe `List` das figuras 21.3–21.5 como a base de uma classe `IndexedList`. Escreva um programa que demonstra a operação de listas indexadas. Certifique-se de incluir as funções-membro `insertInIndexedList`, `searchIndexedList` e `deleteFromIndexedList`.



*A mesma velha e caridosa  
mentira  
Repetida com o passar dos anos  
Perpetuamente é falada —  
Nossa, você realmente não  
mudou nada!*  
Margaret Fishback

*O principal defeito do Rei  
Henrique era mascar pequenos  
pedaços de linha.*  
Hilaire Belloc

*Um texto vigoroso é  
conciso. Uma frase não deve  
conter nenhuma palavra  
desnecessária; um parágrafo,  
nenhuma frase desnecessária.*  
William Strunk, Jr.

## Bits, caracteres, strings C e structs

### OBJETIVOS

Neste capítulo, você aprenderá:

- A criar e utilizar structs.
- A passar structs para funções por valor e por referência.
- Como utilizar typedef para criar aliases (ou sinônimos) de tipos de dados previamente definidos e structs.
- A manipular dados com os operadores de bits e a criar campos de bits para armazenar os dados compactadamente.
- Como utilizar as funções de biblioteca de tratamento de caracteres <cctype>.
- Como utilizar as funções de conversão de strings da biblioteca de utilitários gerais <stdlib>.
- Como utilizar as funções de processamento de string da biblioteca de tratamento de strings <string>.

- 22.1 Introdução
- 22.2 Definições de estrutura
- 22.3 Inicializando estruturas
- 22.4 Utilizando estruturas com funções
- 22.5 typedef
- 22.6 Exemplo: simulação de embaralhamento e distribuição de cartas de alto desempenho
- 22.7 Operadores de bits
- 22.8 Campos de bit
- 22.9 Biblioteca de tratamento de caractere
- 22.10 Funções de conversão de string baseada em ponteiro
- 22.11 Funções de pesquisa da biblioteca de tratamento de strings baseadas em ponteiro
- 22.12 Funções de memória da biblioteca de tratamento de strings baseadas em ponteiro
- 22.13 Síntese

Resumo | Terminologia | Exercícios de revisão | Respostas dos exercícios de revisão | Exercícios

## 22.1 Introdução

Neste capítulo, discutimos estruturas e a manipulação de bits, caracteres e strings no estilo C. Muitas das técnicas que apresentamos aqui são incluídas para o benefício do programador em C++ que trabalhará com C e com os primeiros códigos C++ legados.

Os designers do C++ fizeram as estruturas evoluir para a noção de classe. Semelhantemente a uma classe, as estruturas C++ podem conter especificadores de acesso, funções-membro, construtores e destrutores. De fato, a única diferença entre estruturas e classes em C++ é que os membros de estrutura assumem o padrão de acesso `public` e os membros de classe assumem o padrão de acesso `private` quando especificadores de acesso não são utilizados. As classes foram abrangidas completamente no livro, portanto não há realmente nenhuma necessidade de discutirmos estruturas em detalhes. Nossa apresentação de estruturas neste capítulo focaliza sua utilização em C, em que as estruturas contêm somente membros de dados `public`. Essa utilização de estruturas é típica do código C legado e dos primeiros códigos de C++ que você verá na indústria.

Discutimos como declarar e inicializar estruturas e como passá-las para funções. Em seguida, apresentamos uma simulação de embaralhamento e distribuição de cartas de alto desempenho em que utilizamos objetos de estrutura e strings no estilo C para representar as cartas. Discutimos os operadores de bits que permitem aos programadores acessar e manipular os bits individuais em bytes de dados. Apresentamos também os campos de bit (*bitfields*) — estruturas especiais que podem ser utilizadas para especificar o número exato de bits que uma variável ocupa na memória. Essas técnicas de manipulação de bits são comuns em programas C e C++ que interagem diretamente com dispositivos de hardware com memória limitada. O capítulo termina com exemplos de muitas funções de manipulação de caracteres e strings no estilo C — algumas das quais são projetadas para processar blocos de memória como arrays de bytes.

## 22.2 Definições de estrutura

As estruturas são **tipos de dados agregados** — isto é, eles podem ser construídos utilizando elementos de vários tipos que incluem outros structs. Considere a seguinte definição de estrutura:

```
struct Card
{
 char *face;
 char *suit;
}; // fim do struct Card
```

A palavra-chave **struct** introduz a definição da estrutura Card. O identificador Card é o **nome da estrutura** e é utilizado em C++ para declarar as variáveis do **tipo de estrutura** (no C, o nome de tipo da estrutura precedente é `struct Card`). Neste exemplo, o tipo de estrutura é Card. Os dados (e possivelmente as funções — assim como com as classes) declarados dentro das chaves da definição de estrutura são **membros** da estrutura. Os membros da mesma estrutura devem ter nomes únicos, mas duas estruturas diferentes podem conter membros do mesmo nome sem conflito. Cada definição de estrutura deve terminar com um ponto-e-vírgula.



### Erro comum de programação 22.1

*Esquecer do ponto-e-vírgula que termina uma definição de estrutura é um erro de sintaxe.*

A definição de Card contém dois membros do tipo `char *` — `face` e `suit`. Os membros de estrutura podem ser variáveis dos tipos de dados fundamentais (por exemplo, `int`, `double` etc.) ou agregados, tais como arrays, outras estruturas e/ou classes. Os membros de



dados em uma única definição de estrutura podem ser de muitos tipos de dados. Por exemplo, uma estrutura `Employee` poderia conter membros de string de caracteres para o nome e o sobrenome, um membro `int` para a idade do empregado, um membro `char` contendo 'M' ou 'F' para o gênero do empregado, um membro `double` para o salário hora do empregado e assim por diante.

Uma estrutura não pode conter uma instância de si mesma. Por exemplo, uma variável de estrutura `Card` não pode ser declarada na definição para a estrutura `Card`. Entretanto, um ponteiro para uma estrutura `Card` pode ser incluído. Uma estrutura contendo um membro que é um ponteiro para o mesmo tipo de estrutura é referida como uma **estrutura auto-referencial**. Utilizamos uma construção semelhante — classes auto-referenciais — no Capítulo 21, “Estruturas de dados”, para construir vários tipos de estruturas de dados vinculadas.

A definição de estrutura `Card` não reserva nenhum espaço na memória; em vez disso, cria um novo tipo de dados que é utilizado para declarar variáveis de estrutura. As variáveis de estrutura são declaradas como variáveis de outros tipos. As seguintes declarações

```
Card oneCard;
Card deck[52];
Card *cardPtr;
```

declaram `oneCard` como uma variável de estrutura do tipo `Card`, `deck` como um array de 52 elementos do tipo `Card` e `cardPtr` como um ponteiro para uma estrutura `Card`. As variáveis de um dado tipo de estrutura também podem ser declaradas colocando uma **lista separada por vírgulas** dos nomes de variáveis entre a chave de fechamento da definição de estrutura e o ponto-e-vírgula que termina a definição de estrutura. Por exemplo, as declarações precedentes poderiam ter sido incorporadas na definição de estrutura `Card` assim:

```
struct Card
{
 char *face;
 char *suit;
} oneCard, deck[52], *cardPtr;
```

O nome de estrutura é opcional. Se uma definição de estrutura não contém um nome de estrutura, as variáveis do tipo de estrutura podem ser declaradas somente entre a chave de fechamento direita da definição de estrutura e o ponto-e-vírgula que termina a definição de estrutura.



### Observação de engenharia de software 22.1

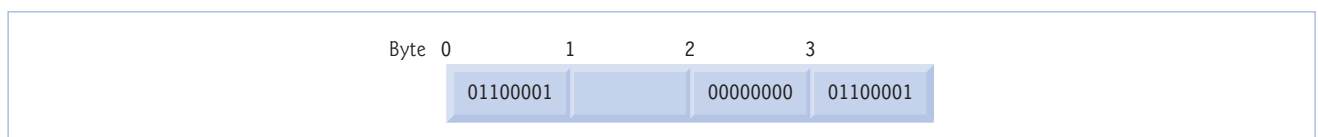
*Forneça um nome de estrutura ao criar um tipo de estrutura. O nome de estrutura é necessário para declarar novas variáveis do tipo de estrutura posteriormente no programa, declarar parâmetros do tipo de estrutura e, se a estrutura estiver sendo utilizada como uma classe C++, especificar o nome do construtor e do destrutor.*

As únicas operações predefinidas válidas que podem ser realizadas em objetos estrutura são: atribuir um objeto estrutura a um objeto estrutura do mesmo tipo, aceitar o endereço (&) de um objeto estrutura, acessar os membros de um objeto estrutura (da mesma maneira que os membros de uma classe são acessados) e utilizar o operador `sizeof` para determinar o tamanho de uma estrutura. Como com as classes, a maioria dos operadores pode ser sobrecarregada para funcionar com objetos de um tipo de estrutura.

Os membros estrutura não são necessariamente armazenados em bytes consecutivos da memória. Às vezes há ‘lacunas’ em uma estrutura, porque alguns computadores armazenam tipos de dados específicos apenas em certos limites de memória, como meia palavra, palavra ou limites de palavra dupla. Uma palavra é uma unidade de memória padrão utilizada para armazenar dados em um computador — normalmente dois ou quatro bytes e, em geral, quatro bytes nos sistemas populares de 32 bits de hoje. Considere a seguinte definição de estrutura em que os objetos de estrutura `sample1` e `sample2` do tipo `Example` são declarados:

```
struct Example
{
 char c;
 int i;
} sample1, sample2;
```

Um computador com palavras de dois bytes poderia requerer que cada um dos membros de `Example` fosse alinhado em um limite de palavra (isto é, no começo de uma palavra — isso é dependente de máquina). A Figura 22.1 mostra um alinhamento de armazenamento de exemplo de um objeto do tipo `Example` que recebeu o caractere 'a' e o inteiro 97 (as representações de bits dos valores são mostradas). Se os membros são armazenados começando em limites de palavra, há uma lacuna de um byte (byte 1 na figura) no armazenamento de objetos do tipo `Example`. O valor na lacuna de 1 byte é indefinido. Se os valores de membro de `sample1` e `sample2` são de fato iguais,



**Figura 22.1** Possível alinhamento de armazenamento para uma variável do tipo `Example`, mostrando uma área indefinida na memória.



os objetos estrutura não serão necessariamente iguais, porque é provável que as lacunas de 1 byte indefinidas não contenham valores idênticos.



### Erro comum de programação 22.2

*Comparar estruturas é um erro de compilação.*



### Dica de portabilidade 22.1

*Como o tamanho de itens de dados de um tipo particular é dependente de máquina, e como as considerações de alinhamento de armazenamento são dependentes de máquinas, a representação de uma estrutura também é dependente de máquina.*

## 22.3 Inicializando estruturas

As estruturas podem ser inicializadas utilizando listas de inicializadores, tal como é feito com arrays. Por exemplo, a declaração

```
Card oneCard = { "Three", "Hearts" };
```

cria a variável `Card oneCard` e inicializa o membro `face` como "Three" e o membro `suit` como "Hearts". Se houver menos inicializadores na lista do que membros na estrutura, os membros restantes são inicializados com seus valores-padrão. As variáveis de estrutura declaradas fora de uma definição de função (isto é, externamente) são inicializadas com seus valores-padrão se não forem explicitamente inicializadas na declaração externa. As variáveis de estrutura também podem ser configuradas em expressões de atribuição atribuindo uma variável de estrutura do mesmo tipo ou atribuindo valores aos membros individuais de dados da estrutura.

## 22.4 Utilizando estruturas com funções

Há duas maneiras de passar as informações em estruturas para as funções. Você pode passar a estrutura inteira ou passar os membros individuais de uma estrutura. Por padrão, as estruturas são passadas por valor. As estruturas e seus membros também podem ser passados por referência passando referências ou ponteiros.

Para passar uma estrutura por referência, passe o endereço do objeto estrutura ou uma referência ao objeto estrutura. Os arrays de estruturas — como todos os outros arrays — são passados por referência.

No Capítulo 7, declaramos que um array poderia ser passado por valor utilizando uma estrutura. Para passar um array por valor, crie uma estrutura (ou uma classe) com o array como um membro, e então passe um objeto desse tipo de estrutura (ou classe) para uma função por valor. Como os objetos de estrutura são passados por valor, o membro de array também é passado por valor.



### Dica de desempenho 22.1

*Passar estruturas (e estruturas especialmente grandes) por referência é mais eficiente que passá-las por valor (o que requer que a estrutura inteira seja copiada).*

## 22.5 typedef

A palavra-chave **typedef** fornece um mecanismo para criar aliases de tipos de dados previamente definidos. Os nomes de tipos de estrutura são frequentemente definidos com **typedef** para criar nomes de tipo mais curtos, mais simples ou mais legíveis. Por exemplo, a instrução

```
typedef Card *CardPtr;
```

define o novo nome de tipo `CardPtr` como um sinônimo do tipo `Card *`.



### Boa prática de programação 22.1

*Coloque em maiúsculas os nomes **typedef** para enfatizar que eles são sinônimos de outros nomes de tipo.*

Criar um novo nome com **typedef** não cria um novo tipo; **typedef** simplesmente cria um novo nome de tipo que pode então ser utilizado no programa como um alias de um nome de tipo existente.



### Dica de portabilidade 22.2

*Os sinônimos para tipos de dados predefinidos podem ser criados com **typedef** para tornar programas mais portáveis. Por exemplo, um programa pode utilizar **typedef** para criar um alias `Integer` para inteiros de quatro bytes. `Integer` pode então ser convertido em um alias de `int`, em sistemas com inteiros de quatro bytes, e um alias `long int` em sistemas com inteiros de dois bytes em que valores `long int` ocupam quatro bytes. Então, o programador simplesmente declara todas as variáveis inteiras de quatro bytes como sendo do tipo `Integer`.*

## 22.6 Exemplo: simulação de embaralhamento e distribuição de cartas de alto desempenho

O programa nas figuras 22.2–22.4 é baseado na simulação de embaralhamento e distribuição de cartas discutida no Capítulo 8. O programa representa as cartas do baralho como um array de estruturas e utiliza os algoritmos de embaralhamento e distribuição de alto desempenho.

No programa, o construtor inicializa o array `Card` pela ordem das strings de caracteres que representam as cartas de Ás a Rei de cada naipe. A função `shuffle` é onde o algoritmo de embaralhamento de alto desempenho é implementado. A função faz um loop por todas as 52 cartas (subscritos de array 0 a 51). Para cada carta, um número entre 0 e 51 é selecionado aleatoriamente. Em seguida, a estrutura `Card` atual e a estrutura `Card` selecionada aleatoriamente são trocadas no array. Um total de 52 permutações é feito em uma única passagem pelo array inteiro, e o array de estruturas `Card` é embaralhado! Diferentemente do algoritmo de embaralhamento apresentado no Capítulo 8, esse algoritmo não sofre adiamento indefinido. Como as estruturas `Card` foram permutadas ‘no local’ no array, o algoritmo de distribuição de cartas de alto desempenho implementado na função `deal` requer somente uma passagem do array para distribuir as cartas embaralhadas.

```

1 // Figura 22.2: DeckOfCards.h
2 // Definição da classe DeckOfCards que
3 // representa um baralho.
4
5 // Definição da estrutura Card
6 struct Card
7 {
8 char *face;
9 char *suit;
10 }; // fim da estrutura Card
11
12 // Definição da classe DeckOfCards
13 class DeckOfCards
14 {
15 public:
16 DeckOfCards(); // construtor inicializa deck
17 void shuffle(); // embaralha as cartas do baralho
18 void deal(); // distribui as cartas do baralho
19
20 private:
21 Card deck[52]; // representa o baralho de cartas
22 }; // fim da classe DeckOfCards

```

**Figura 22.2** Arquivo de cabeçalho da classe `DeckOfCards`.

```

1 // Figura 22.3: DeckOfCards.cpp
2 // Definições de função-membro para a classe DeckOfCards que simula
3 // o embaralhamento e distribuição de um baralho.
4 #include <iostream>
5 using std::cout;
6 using std::left;
7 using std::right;
8
9 #include <iomanip>
10 using std::setw;
11
12 #include <cstdlib> // protótipos para rand e srand
13 using std::rand;

```

**Figura 22.3** Arquivo de classe para `DeckOfCards`.

(continua)

```
14 using std::srand;
15
16 #include <ctime> // protótipo para time
17 using std::time;
18
19 #include "DeckOfCards.h" // definição da classe DeckOfCards
20
21 // construtor DeckOfCards sem argumento inicializa o baralho
22 DeckOfCards::DeckOfCards()
23 {
24 // inicializa o array suit
25 static char *suit[4] =
26 { "Hearts", "Diamonds", "Clubs", "Spades" };
27
28 // inicializa o array face
29 static char *face[13] =
30 { "Ace", "Deuce", "Three", "Four", "Five", "Six", "Seven",
31 "Eight", "Nine", "Ten", "Jack", "Queen", "King" };
32
33 // configura valores para o baralho de 52 cartas
34 for (int i = 0; i < 52; i++)
35 {
36 deck[i].face = face[i % 13];
37 deck[i].suit = suit[i / 13];
38 } // fim do for
39
40 srand(time(0)); // semeia o gerador de número aleatório
41 } // fim do construtor DeckOfCards sem argumento
42
43 // embaralha as cartas do baralho
44 void DeckOfCards::shuffle()
45 {
46 // embaralha cartas aleatoriamente
47 for (int i = 0; i < 52; i++)
48 {
49 int j = rand() % 52;
50 Card temp = deck[i];
51 deck[i] = deck[j];
52 deck[j] = temp;
53 } // fim do for
54 } // fim da função shuffle
55
56 // distribui as cartas do baralho
57 void DeckOfCards::deal()
58 {
59 // exibe a face e o naipe de cada carta
60 for (int i = 0; i < 52; i++)
61 cout << right << setw(5) << deck[i].face << " of "
62 << left << setw(8) << deck[i].suit
63 << ((i + 1) % 2 ? '\t' : '\n');
64 } // fim da função deal
```

Figura 22.3 Arquivo de classe para DeckOfCards.

(continuação)

```

1 // Figura 22.4: fig22_04.cpp
2 // Programa de embaralhamento e distribuição de cartas.
3 #include "DeckOfCards.h" // Definição da classe DeckOfCards
4
5 int main()
6 {
7 DeckOfCards deckOfCards; // cria objeto DeckOfCards
8
9 deckOfCards.shuffle(); // embaralha as cartas
10 deckOfCards.deal(); // distribui as cartas
11 return 0; // indica terminação bem-sucedida
12 } // fim do main

```

King of Clubs	Ten of Diamonds
Five of Diamonds	Jack of Clubs
Seven of Spades	Five of Clubs
Three of Spades	King of Hearts
Ten of Clubs	Eight of Spades
Eight of Hearts	Six of Hearts
Nine of Diamonds	Nine of Clubs
Three of Diamonds	Queen of Hearts
Six of Clubs	Seven of Hearts
Seven of Diamonds	Jack of Diamonds
Jack of Spades	King of Diamonds
Deuce of Diamonds	Four of Clubs
Three of Clubs	Five of Hearts
Eight of Clubs	Ace of Hearts
Deuce of Spades	Ace of Clubs
Ten of Spades	Eight of Diamonds
Ten of Hearts	Six of Spades
Queen of Diamonds	Nine of Hearts
Seven of Clubs	Queen of Clubs
Deuce of Clubs	Queen of Spades
Three of Hearts	Five of Spades
Deuce of Hearts	Jack of Hearts
Four of Hearts	Ace of Diamonds
Nine of Spades	Four of Diamonds
Ace of Spades	Six of Diamonds
Four of Spades	King of Spades

**Figura 22.4** Simulação do embaralhamento e distribuição de cartas de alto desempenho.

## 22.7 Operadores de bits

O C++ fornece extensas capacidades de manipulação de bits para os programadores que precisam descer ao chamado nível dos ‘bits e bytes’. Sistemas operacionais, software de equipamento de testes, software de rede e muitos outros tipos de software exigem que o programador se comunique ‘diretamente com o hardware’. Nesta e nas próximas seções, discutimos as capacidades de manipulação de bits do C++. Introduzimos cada um dos muitos operadores de bits do C++ e como economizar memória utilizando campos de bits.

Todos os dados são representados internamente pelos computadores como seqüências de bits. Cada bit pode assumir o valor 0 ou o valor 1. Na maioria dos sistemas, uma seqüência de 8 bits forma um **byte** — a unidade de armazenamento padrão para uma variável do tipo `char`. Outros tipos de dados são armazenados em números de bytes maiores. Os operadores de bits são utilizados para manipular os bits de operandos integrais (`char`, `short`, `int` e `long`; tanto `signed` como `unsigned`). Inteiros sem sinal são normalmente utilizados com os operadores de bits.



### Dica de portabilidade 22.3

*As manipulações de dados de bit a bit são dependentes de máquina.*

Observe que as discussões de operador de bits nesta seção mostram as representações binárias dos operandos de inteiro. Para uma explicação detalhada do sistema de números binários (também chamado de base 2), consulte Apêndice D, “Sistemas de numeração”. Por causa da natureza dependente de máquina das manipulações de bits, alguns desses programas talvez não funcionem em seu sistema sem uma modificação.

Os operadores de bits são: **E sobre bits (&)**, **OU inclusivo sobre bits (|)**, **OU exclusivo sobre bits (^)**, **deslocamento para a esquerda (<<)**, **deslocamento para a direita (>>)** e **complemento de bits (~)** — também conhecido como **complemento de um**. (Observe que temos utilizado &, << e >> para outros propósitos. Esse é um exemplo clássico de sobrecarga de operadores.) Os operadores E sobre bits, OU inclusivo sobre bits e OU exclusivo sobre bits comparam seus dois operandos bit a bit. O operador E sobre bits configura cada bit no resultado como 1 se o bit correspondente em ambos os operandos for 1. O operador OU inclusivo sobre bits configura cada bit no resultado como 1 se o bit correspondente em qualquer um dos (ou ambos os) operandos for 1. O operador OU exclusivo sobre bits configura cada bit no resultado como 1 se o bit correspondente em qualquer um dos operandos — mas não em ambos — for 1. O operador de deslocamento para a esquerda desloca os bits de seu operando esquerdo para a esquerda pelo número de bits especificado em seu operando direito. O operador de deslocamento para a direita desloca os bits em seu operando esquerdo para a direita pelo número de bits especificado em seu operando direito. O operador de complemento de bits configura todos os bits 0 em seu operando como 1 no resultado e configura todos os bits 1 em seu operando como 0 no resultado. Discussões detalhadas sobre cada operador de bits aparecem nos exemplos a seguir. Os operadores de bits são resumidos na Figura 22.5.

### *Imprimindo uma representação binária de um valor integral*

Ao utilizar os operadores de bits, é útil ilustrar seus efeitos precisos imprimindo valores em sua representação binária. O programa da Figura 22.6 imprime um inteiro `unsigned` em sua representação binária em grupos de oito bits cada.

Operador	Nome	Descrição
&	E sobre bits	Os bits no resultado são configurados como 1 se os bits correspondentes nos dois operandos forem ambos 1.
	OU inclusivo sobre bits	Os bits no resultado são configurados como 1 se um ou ambos os bits correspondentes nos dois operandos for 1.
^	OU exclusivo sobre bits	Os bits no resultado são configurados como 1 se exatamente um dos bits correspondentes nos dois operandos for 1.
<<	deslocamento de bits para a esquerda	Desloca os bits do primeiro operando para a esquerda pelo número de bits especificado pelo segundo operando; preenche a partir da direita com bits 0.
>>	deslocamento para a direita com extensão de sinal	Desloca os bits do primeiro operando direito pelo número de bits especificado pelo segundo operando; o método de preenchimento a partir da esquerda é dependente de máquina.
~	complemento de bits	Todos os bits 0 são configurados como 1 e todos os bits 1 são configurados como 0.

**Figura 22.5** Operadores de bits.

```

1 // Figura 22.6: fig22_06.cpp
2 // Imprimindo um inteiro sem sinal em bits.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 #include <iomanip>
9 using std::setw;
10
11 void displayBits(unsigned); // protótipo
12
13 int main()
14 {

```

**Figura 22.6** Imprimindo um inteiro sem sinal em bits.

(continua)

```

15 unsigned inputValue; // valor integral para imprimir em binário
16
17 cout << "Enter an unsigned integer: ";
18 cin >> inputValue;
19 displayBits(inputValue);
20 return 0;
21 } // fim do main
22
23 // exibe bits de um valor inteiro sem sinal
24 void displayBits(unsigned value)
25 {
26 const int SHIFT = 8 * sizeof(unsigned) - 1;
27 const unsigned MASK = 1 << SHIFT;
28
29 cout << setw(10) << value << " = ";
30
31 // exibe bits
32 for (unsigned i = 1; i <= SHIFT + 1; i++)
33 {
34 cout << (value & MASK ? '1' : '0');
35 value <<= 1; // desloca valor esquerdo por 1
36
37 if (i % 8 == 0) // gera saída de um espaço depois de 8 bits
38 cout << ' ';
39 } // fim do for
40
41 cout << endl;
42 } // fim da função displayBits

```

```

Enter an unsigned integer: 65000
65000 = 00000000 00000000 11111101 11101000

```

```

Enter an unsigned integer: 29
29 = 00000000 00000000 00000000 00011101

```

Figura 22.6 Imprimindo um inteiro sem sinal em bits.

(continuação)

A função `displayBits` (linhas 24–42) utiliza o operador E sobre bits para combinar a variável `value` com a constante `MASK`. O operador E sobre bits costuma ser utilizado com um operando chamado de **máscara** — um valor inteiro com bits específicos configurados como 1. As máscaras são utilizadas para ocultar alguns bits em um valor enquanto se selecionam outros bits. Em `displayBits`, a linha 27 atribui à constante `MASK` o valor `1 << SHIFT`. O valor da constante `SHIFT` foi calculado na linha 26 com a expressão

```
8 * sizeof(unsigned) - 1
```

que multiplica o número de bytes que um objeto `unsigned` requer na memória por 8 (o número de bits em um byte) para obter o número total de bits necessário para armazenar um objeto `unsigned`, então subtrai 1. A representação de bits de `1 << SHIFT` em um computador que representa objetos `unsigned` em quatro bytes de memória é

```
10000000 00000000 00000000 00000000
```

O operador de deslocamento para a esquerda desloca o valor 1 a partir do bit de ordem inferior (mais à direita) para o bit de ordem superior (mais à esquerda) em `MASK` e preenche os bits 0 a partir da direita. A linha 34 determina se um 1 ou um 0 deve ser impresso para o bit mais à esquerda atual da variável `value`. Suponha que a variável `value` contém 65000 (00000000 00000000 11111101 11101000). Quando `value` e `MASK` são combinados utilizando `&`, todos os bits, exceto o bit de ordem superior na variável `value`, são ‘mascarados’ (ocultos), porque qualquer bit submetido ao operador E com 0 produz 0. Se o bit mais à esquerda for 1, `value & MASK` é avaliado como

```

00000000 00000000 11111101 11101000 (value)
10000000 00000000 00000000 00000000 (MASK)

00000000 00000000 00000000 00000000 (value & MASK)

```

que é interpretado como false e 0 é impresso. Então a linha 35 desloca a variável `value` para a esquerda por um bit com a expressão `value <<= 1` (isto é, `value = value << 1`). Esses passos são repetidos para cada variável de bit `value`. Por fim, um bit com um valor de 1 é deslocado na posição de bit mais à esquerda e a manipulação de bits é da seguinte maneira:

```
11111101 11101000 00000000 00000000 (value)
10000000 00000000 00000000 00000000 (MASK)

10000000 00000000 00000000 00000000 (value & MASK)
```

Como ambos os bits esquerdos são 1s, o resultado da expressão é não-zero (verdadeiro) e um valor de 1 é impresso. A Figura 22.7 resume os resultados da combinação de dois bits com o operador E sobre bits.



### Erro comum de programação 22.3

*Utilizar o operador lógico (&&) para o operador E sobre bits (&) e vice-versa é um erro de lógica.*

O programa da Figura 22.8 demonstra o operador E sobre bits, o operador OU inclusivo sobre bits, o operador OU exclusivo sobre bits e o operador de complemento sobre bits. A função `displayBits` (linhas 57–75) imprime os valores de inteiro `unsigned`.

#### Operador E sobre bits (&)

Na Figura 22.8, a linha 21 atribui 2179876355 (10000001 11101110 01000110 00000011) à variável `number1`, e a linha 22 atribui 1 (00000000 00000000 00000000 00000001) à variável `mask`. Quando `mask` e `number1` são combinados utilizando o operador E sobre bits (&) na expressão `number1 & mask` (linha 27), o resultado é 00000000 00000000 00000000 00000001. Todos os bits, exceto o bit de ordem inferior na variável `number1`, são ‘mascarados’ (ocultos) aplicando uma operação ‘E’ com a constante `MASK`.

Bit 1	Bit 2	Bit 1 & bit 2
0	0	0
1	0	0
0	1	0
1	1	1

**Figura 22.7** Resultados da combinação de dois bits com o operador E sobre bits (&).

```
1 // Figura 22.8: fig22_08.cpp
2 // Utilizando os operadores E sobre bits, OU inclusivo E sobre bits
3 // OU exclusivo sobre bits e complemento de bits.
4 #include <iostream>
5 using std::cout;
6
7 #include <iomanip>
8 using std::endl;
9 using std::setw;
10
11 void displayBits(unsigned); // protótipo
12
13 int main()
14 {
15 unsigned number1;
16 unsigned number2;
17 unsigned mask;
18 unsigned setBits;
19
20 // demonstra & sobre bits
```

**Figura 22.8** Operadores E sobre bits, OU inclusivo sobre bits, OU exclusivo sobre bits e de complemento de bits.

(continua)



```

21 number1 = 2179876355;
22 mask = 1;
23 cout << "The result of combining the following\n";
24 displayBits(number1);
25 displayBits(mask);
26 cout << "using the bitwise AND operator & is\n";
27 displayBits(number1 & mask);
28
29 // demonstra | sobre bits
30 number1 = 15;
31 setBits = 241;
32 cout << "\nThe result of combining the following\n";
33 displayBits(number1);
34 displayBits(setBits);
35 cout << "using the bitwise inclusive OR operator | is\n";
36 displayBits(number1 | setBits);
37
38 // demonstra OU exclusivo sobre bits
39 number1 = 139;
40 number2 = 199;
41 cout << "\nThe result of combining the following\n";
42 displayBits(number1);
43 displayBits(number2);
44 cout << "using the bitwise exclusive OR operator ^ is\n";
45 displayBits(number1 ^ number2);
46
47 // demonstra complemento de bits
48 number1 = 21845;
49 cout << "\nThe one's complement of\n";
50 displayBits(number1);
51 cout << "is" << endl;
52 displayBits(~number1);
53 return 0;
54 } // fim do main
55
56 // exibe bits de um valor inteiro sem sinal
57 void displayBits(unsigned value)
58 {
59 const int SHIFT = 8 * sizeof(unsigned) - 1;
60 const unsigned MASK = 1 << SHIFT;
61
62 cout << setw(10) << value << " = ";
63
64 // exibe bits
65 for (unsigned i = 1; i <= SHIFT + 1; i++)
66 {
67 cout << (value & MASK ? '1' : '0');
68 value <<= 1; // desloca valor esquerdo por 1
69
70 if (i % 8 == 0) // gera saída de um espaço depois de 8 bits
71 cout << ' ';
72 } // fim do for
73
74 cout << endl;
75 } // fim da função displayBits

```

**Figura 22.8** Operadores E sobre bits, OU inclusivo sobre bits, OU exclusivo sobre bits e de complemento de bits.

(continua)

```

The result of combining the following
2179876355 = 10000001 11101110 01000110 00000011
1 = 00000000 00000000 00000000 00000001
using the bitwise AND operator & is
1 = 00000000 00000000 00000000 00000001

The result of combining the following
15 = 00000000 00000000 00000000 00001111
241 = 00000000 00000000 00000000 11110001
using the bitwise inclusive OR operator | is
255 = 00000000 00000000 00000000 11111111

The result of combining the following
139 = 00000000 00000000 00000000 10001011
199 = 00000000 00000000 00000000 11000111
using the bitwise exclusive OR operator ^ is
76 = 00000000 00000000 00000000 01001100

The one's complement of
21845 = 00000000 00000000 01010101 01010101
is
4294945450 = 11111111 11111111 10101010 10101010

```

**Figura 22.8** Operadores E sobre bits, OU inclusivo sobre bits, OU exclusivo sobre bits e de complemento de bits.

(continuação)

### Operador OU inclusivo sobre bits (|)

O operador OU inclusivo sobre bits é utilizado para configurar bits específicos como 1 em um operando. Na Figura 22.8, a linha 30 atribui 15 (00000000 00000000 00000000 00001111) à variável `number1` e a linha 31 atribui 241 (00000000 00000000 00000000 11110001) à variável `setBits`. Quando `number1` e `setBits` são combinados utilizando o operador OU sobre bits na expressão `number1 | setBits` (linha 36), o resultado é 255 (00000000 00000000 00000000 11111111). A Figura 22.9 resume os resultados da combinação de dois bits com o operador OU inclusivo sobre bits.



### Erro comum de programação 22.4

Utilizar o operador lógico (||) para o operador OU sobre bits (|) e vice-versa é um erro de lógica.

### OU exclusivo sobre bits (^)

O operador OU exclusivo sobre bits (^) configura cada bit no resultado como 1 se *exatamente* um dos bits correspondentes em seus dois operandos for 1. Na Figura 22.8, as linhas 39–40 atribuem às variáveis `number1` e `number2` os valores 139 (00000000 00000000 00000000 10001011) e 199 (00000000 00000000 00000000 11000111), respectivamente. Quando essas variáveis são combinadas com o operador OU exclusivo na expressão `number1 ^ number2` (linha 45), o resultado é 76 (00000000 00000000 00000000 01001100). A Figura 22.10 resume resultados da combinação de dois bits com o operador OU exclusivo sobre bits.

### Complemento de bits (~)

O operador de complemento de bits (~) configura todos os bits 1 em seu operando como 0 no resultado e configura todos os bits 0 como 1 no resultado — o que também é referido como ‘obter o complemento de um do valor’. Na Figura 22.8, a linha 48 atribui à variável `number121845` (00000000 00000000 01010101 01010101). Quando a expressão `~number1` é avaliada, o resultado é (11111111 11111111 10101010 10101010).

A Figura 22.11 demonstra o operador de deslocamento para a esquerda (<<) e o de deslocamento para a direita (>>). A função `displayBits` (linhas 31–49) imprime os valores de inteiro unsigned.

### Operador de deslocamento para a esquerda

O operador de deslocamento para a esquerda (<<) desloca os bits de seu operando esquerdo para a esquerda pelo número de bits especificado em seu operando direito. Os bits tornados vagos à direita são substituídos por 0s; bits deslocados para fora à esquerda são perdidos. No programa da Figura 22.11, a linha 14 atribui à variável `number1` valor 960 (00000000 00000000 00000011 11000000). O resultado do deslocamento para a esquerda da variável `number1` 8 bits na expressão `number1 << 8` (linha 20) é 245760 (00000000 00000011 11000000 00000000).

Bit 1	Bit 2	Bit 1   bit 2
0	0	0
1	0	1
0	1	1
1	1	1

**Figura 22.9** Combinando dois bits com o operador OU inclusivo sobre bits (|).

Bit 1	Bit 2	Bit 1 ^ bit 2
0	0	0
1	0	1
0	1	1
1	1	0

**Figura 22.10** Combinando dois bits com o operador OU exclusivo sobre bits (^).

```

1 // Figura 22.11: fig22_11.cpp
2 // Utilizando os operadores de deslocamento de bits.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 void displayBits(unsigned); // protótipo
11
12 int main()
13 {
14 unsigned number1 = 960;
15
16 // demonstra deslocamento de bits para a esquerda
17 cout << "The result of left shifting\n";
18 displayBits(number1);
19 cout << "8 bit positions using the left-shift operator is\n";
20 displayBits(number1 << 8);
21
22 // demonstra deslocamento de bits para a direita
23 cout << "\nThe result of right shifting\n";
24 displayBits(number1);
25 cout << "8 bit positions using the right-shift operator is\n";
26 displayBits(number1 >> 8);
27 return 0;
28 } // fim do main
29
30 // exibe bits de um valor inteiro sem sinal
31 void displayBits(unsigned value)
32 {
33 const int SHIFT = 8 * sizeof(unsigned) - 1;

```

**Figura 22.11** Os operadores de deslocamento de bits.

(continua)

```

34 const unsigned MASK = 1 << SHIFT;
35
36 cout << setw(10) << value << " = ";
37
38 // exibe bits
39 for (unsigned i = 1; i <= SHIFT + 1; i++)
40 {
41 cout << (value & MASK ? '1' : '0');
42 value <<= 1; // desloca valor esquerdo por 1
43
44 if (i % 8 == 0) // gera saída de um espaço depois de 8 bits
45 cout << ' ';
46 } // fim do for
47
48 cout << endl;
49 } // fim da função displayBits

```

The result of left shifting  
 960 = 00000000 00000000 00000011 11000000  
 8 bit positions using the left-shift operator is  
 245760 = 00000000 00000011 11000000 00000000

The result of right shifting  
 960 = 00000000 00000000 00000011 11000000  
 8 bit positions using the right-shift operator is  
 3 = 00000000 00000000 00000000 00000011

**Figura 22.11** Os operadores de deslocamento de bits.

(continuação)

### Operador de deslocamento para a direita

O operador de deslocamento para a direita (>>) desloca os bits de seu operando esquerdo para a direita pelo número de bits especificado em seu operando direito. Realizar um deslocamento para a direita em um inteiro `unsigned` faz com que os bits tornados vagos à esquerda sejam substituídos por 0s; os bits deslocados para fora à direita são perdidos. No programa da Figura 22.11, o resultado de deslocar `number1` para a direita na expressão `number1 >> 8` (linha 26) é 3 (00000000 00000000 00000000 00000011).



### Erro comum de programação 22.5

*O resultado de deslocar um valor é indefinido se o operando direito for negativo ou se o operando direito for maior que ou igual ao número de bits em que o operando esquerdo é armazenado.*



### Dica de portabilidade 22.4

*O resultado de deslocamento para a direita de um valor com sinal é dependente de máquina. Algumas máquinas preenchem com zeros e outras utilizam o bit de sinal.*

### Operadores de atribuição de bits

Cada operador de bits (exceto o operador de complemento de bits) tem um operador de atribuição correspondente. Esses *operadores de atribuição de bits* são mostrados na Figura 22.12 e são utilizados de uma maneira semelhante aos operadores de atribuição aritméticos introduzidos no Capítulo 2.

A Figura 22.13 mostra a precedência e a associatividade dos operadores introduzidos até agora no texto. Eles são mostrados de cima para baixo em ordem decrescente de precedência.

## 22.8 Campos de bit

O C++ fornece a capacidade de especificar o número de bits em que um membro do tipo inteiro ou do tipo `enum` de uma classe ou uma estrutura é armazenado. Tal membro é referido como **campo de bit**. Os campos de bit permitem melhor utilização de memória armazenando dados no número mínimo de bits requerido. Os membros de campo de bit *devem* ser declarados como um tipo inteiro ou `enum`.

Operadores de atribuição de bits	
<code>&amp;=</code>	Operador de atribuição E sobre bits.
<code> =</code>	Operador de atribuição OU inclusivo sobre bits.
<code>^=</code>	Operador de atribuição OU exclusivo sobre bits.
<code>&lt;&lt;=</code>	Operador de atribuição de deslocamento para a esquerda.
<code>&gt;&gt;=</code>	Deslocamento para a direita com operador de atribuição com extensão de sinal.

Figura 22.12 Operadores de atribuição de bits.

Operadores	Associatividade	Tipo
<code>::</code> (unário; da direita para a esquerda) <code>::</code> (binário; da esquerda para a direita)	da esquerda para a direita	mais alto
<code>() [] . -&gt; ++ -- static_cast&lt;tipo&gt;()</code>	da esquerda para a direita	unário
<code>++ -- + - ! delete sizeof</code>	da direita para a esquerda	unário
<code>* ~ &amp; new</code>		
<code>* / %</code>	da esquerda para a direita	multiplicativo
<code>+ -</code>	da esquerda para a direita	aditivo
<code>&lt;&lt; &gt;&gt;</code>	da esquerda para a direita	deslocamento
<code>&lt; &lt;= &gt; &gt;=</code>	da esquerda para a direita	relacional
<code>== !=</code>	da esquerda para a direita	igualdade
<code>&amp;</code>	da esquerda para a direita	E sobre bits
<code>^</code>	da esquerda para a direita	XOU sobre bits
<code> </code>	da esquerda para a direita	OU sobre bits
<code>&amp;&amp;</code>	da esquerda para a direita	E lógico
<code>  </code>	da esquerda para a direita	OU lógico
<code>?:</code>	da direita para a esquerda	ternário condicional
<code>= += -= *= /= %= &amp;=  = ^= &lt;&lt;= &gt;&gt;=</code>	da direita para a esquerda	atribuição
<code>,</code>	da esquerda para a direita	vírgula

Figura 22.13 Precedência e associatividade de operadores.



## Dica de desempenho 22.2

*Os campos de bit ajudam a conservar o armazenamento.*

Considere a seguinte definição de estrutura:

```
struct BitCard
{
 unsigned face : 4;
 unsigned suit : 2;
 unsigned color : 1;
}; // fim do struct BitCard
```

A definição contém três campos de bit unsigned — face, suit e color — utilizados para representar um baralho de 52 cartas. Um campo de bit é declarado colocando-se depois de um membro do tipo inteiro ou do tipo enum dois-pontos (:) e uma constante do tipo

inteiro representando a **largura do campo de bit** (isto é, o número de bits em que o membro é armazenado). A largura deve ser uma constante do tipo inteiro.

A definição precedente de estrutura indica que o membro `face` é armazenado em 4 bits, o membro `suit` em 2 bits e o membro `color` em 1 bit. O número de bits é baseado no intervalo desejado de valores para cada membro de estrutura. O membro `face` armazena valores entre 0 (Ace) e 12 (King) — 4 bits podem armazenar um valor entre 0 e 15. O membro `suit` armazena valores entre 0 e 3 (0 = Diamonds [Ouros], 1 = Hearts [Copas], 2 = Clubs [Paus], 3 = Spades [Espada]) — 2 bits podem armazenar um valor entre 0 e 3. Por fim, o membro `color` armazena 0 (Red [Vermelho]) ou 1 (Black [Preto]) — 1 bit pode armazenar 0 ou 1.

O programa nas figuras 22.14–22.16 cria o array `deck` contendo 52 estruturas `BitCard` (linha 21 da Figura 22.14). O construtor insere as 52 cartas no array `deck` e a função `deal` imprime as 52 cartas. Note que os campos de bit são acessados exatamente como qualquer outro membro de estrutura (linhas 18–20 e 28–33 da Figura 22.15). O membro `color` é incluído como um meio de indicar a cor da carta em um sistema que permite monitores coloridos.

```

1 // Figura 22.14: DeckOfCards.h
2 // Definição da classe DeckOfCards que
3 // representa um baralho.
4
5 // Definição de estrutura BitCard com campos de bit
6 struct BitCard
7 {
8 unsigned face : 4; // 4 bits; 0-15
9 unsigned suit : 2; // 2 bits; 0-3
10 unsigned color : 1; // 1 bit; 0-1
11 }; // fim do struct BitCard
12
13 // definição da classe DeckOfCards
14 class DeckOfCards
15 {
16 public:
17 DeckOfCards(); // construtor inicializa deck
18 void deal(); // distribui as cartas do baralho
19
20 private:
21 BitCard deck[52]; // representa o baralho de cartas
22 }; // fim da classe DeckOfCards

```

**Figura 22.14** O arquivo de cabeçalho para a classe `DeckOfCards`.

```

1 // Figura 22.15: DeckOfCards.cpp
2 // Definições de função-membro para a classe DeckOfCards que simula
3 // o embaralhamento e a distribuição de um baralho.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include <iomanip>
9 using std::setw;
10
11 #include "DeckOfCards.h" // definição da classe DeckOfCards
12
13 // construtor DeckOfCards sem argumento inicializa o baralho
14 DeckOfCards::DeckOfCards()
15 {

```

**Figura 22.15** Arquivo de classe para `DeckOfCards`.

(continua)

```

16 for (int i = 0; i <= 51; i++)
17 {
18 deck[i].face = i % 13; // faces na ordem
19 deck[i].suit = i / 13; // naipes na ordem
20 deck[i].color = i / 26; // cores na ordem
21 } // fim do for
22 } // fim do construtor DeckOfCards sem argumento
23
24 // distribui as cartas do baralho
25 void DeckOfCards::deal()
26 {
27 for (int k1 = 0, k2 = k1 + 26; k1 <= 25; k1++, k2++)
28 cout << "Card:" << setw(3) << deck[k1].face
29 << " Suit:" << setw(2) << deck[k1].suit
30 << " Color:" << setw(2) << deck[k1].color
31 << " " << "Card:" << setw(3) << deck[k2].face
32 << " Suit:" << setw(2) << deck[k2].suit
33 << " Color:" << setw(2) << deck[k2].color << endl;
34 } // fim da função deal

```

Figura 22.15 Arquivo de classe para DeckOfCards.

(continuação)

É possível especificar um **campo de bit não identificado**, caso em que o campo é utilizado como **preenchimento** na estrutura. Por exemplo, a definição de estrutura utiliza um campo de 3 bits como preenchimento — nada pode ser armazenado nesses 3 bits. O membro *b* é armazenado em outra unidade de armazenamento.

```

struct Example
{
 unsigned a : 13;
 unsigned : 3; // alinha com o próximo limite de unidade de armazenamento
 unsigned b : 4;
}; // fim do struct Example

```

Um **campo de bit não identificado com uma largura de zero** é utilizado para alinhar o próximo campo de bit em um novo limite de unidade de armazenamento. Por exemplo, a definição de estrutura

```

struct Example
{
 unsigned a : 13;
 unsigned : 0; // alinha com o próximo limite de unidade de armazenamento
 unsigned b : 4;
}; // fim do struct Example

```

utiliza um campo de 0 bit não identificado para pular os bits restantes (quantos houver) da unidade de armazenamento em que *a* é armazenado e alinha *b* no próximo limite de unidade de armazenamento.



### Dica de portabilidade 22.5

As manipulações de campo de bits são dependentes de máquina. Por exemplo, alguns computadores permitem aos campos de bit cruzar limites de palavra, enquanto outros não permitem.



### Erro comum de programação 22.6

Tentar acessar bits individuais de um campo de bit com subscrito como se eles fossem elementos de um array é um erro de compilação. Os campos de bit não são ‘arrays de bit’.



### Erro comum de programação 22.7

Tentar obter o endereço de um campo de bit (o operador & pode não ser utilizado com campos de bit porque um ponteiro pode designar somente um byte particular na memória e os campos de bit podem iniciar no meio de um byte) é um erro de compilação.



```

1 // Figura 22.16: fig22_16.cpp
2 // Programa de embaralhamento e distribuição de cartas.
3 #include "DeckOfCards.h" // definição da classe DeckOfCards
4
5 int main()
6 {
7 DeckOfCards deckOfCards; // cria objeto DeckOfCards
8 deckOfCards.deal(); // distribui as cartas
9 return 0; // indica terminação bem-sucedida
10 } // fim do main

```

```

Card: 0 Suit: 0 Color: 0 Card: 0 Suit: 2 Color: 1
Card: 1 Suit: 0 Color: 0 Card: 1 Suit: 2 Color: 1
Card: 2 Suit: 0 Color: 0 Card: 2 Suit: 2 Color: 1
Card: 3 Suit: 0 Color: 0 Card: 3 Suit: 2 Color: 1
Card: 4 Suit: 0 Color: 0 Card: 4 Suit: 2 Color: 1
Card: 5 Suit: 0 Color: 0 Card: 5 Suit: 2 Color: 1
Card: 6 Suit: 0 Color: 0 Card: 6 Suit: 2 Color: 1
Card: 7 Suit: 0 Color: 0 Card: 7 Suit: 2 Color: 1
Card: 8 Suit: 0 Color: 0 Card: 8 Suit: 2 Color: 1
Card: 9 Suit: 0 Color: 0 Card: 9 Suit: 2 Color: 1
Card: 10 Suit: 0 Color: 0 Card: 10 Suit: 2 Color: 1
Card: 11 Suit: 0 Color: 0 Card: 11 Suit: 2 Color: 1
Card: 12 Suit: 0 Color: 0 Card: 12 Suit: 2 Color: 1
Card: 0 Suit: 1 Color: 0 Card: 0 Suit: 3 Color: 1
Card: 1 Suit: 1 Color: 0 Card: 1 Suit: 3 Color: 1
Card: 2 Suit: 1 Color: 0 Card: 2 Suit: 3 Color: 1
Card: 3 Suit: 1 Color: 0 Card: 3 Suit: 3 Color: 1
Card: 4 Suit: 1 Color: 0 Card: 4 Suit: 3 Color: 1
Card: 5 Suit: 1 Color: 0 Card: 5 Suit: 3 Color: 1
Card: 6 Suit: 1 Color: 0 Card: 6 Suit: 3 Color: 1
Card: 7 Suit: 1 Color: 0 Card: 7 Suit: 3 Color: 1
Card: 8 Suit: 1 Color: 0 Card: 8 Suit: 3 Color: 1
Card: 9 Suit: 1 Color: 0 Card: 9 Suit: 3 Color: 1
Card: 10 Suit: 1 Color: 0 Card: 10 Suit: 3 Color: 1
Card: 11 Suit: 1 Color: 0 Card: 11 Suit: 3 Color: 1
Card: 12 Suit: 1 Color: 0 Card: 12 Suit: 3 Color: 1

```

**Figura 22.16** Os campos de bit utilizados para armazenar um baralho de cartas.



### Dica de desempenho 22.3

*Embora os campos de bit economizem espaço, utilizá-los pode fazer com que o compilador gere código de linguagem de máquina de execução mais lenta. Isso ocorre porque ele aceita operações extras de linguagem de máquina para acessar somente partes de uma unidade de armazenamento endereçável. Esse é um dos muitos exemplos das relações de troca entre espaço e tempo que ocorrem na ciência da computação.*

## 22.9 Biblioteca de tratamento de caractere

A maioria dos dados é inserida em computadores como caracteres — incluindo letras, dígitos e vários símbolos especiais. Nesta seção, discutimos as capacidades do C++ de examinar e manipular caracteres individuais. No restante do capítulo, continuamos nossa discussão sobre manipulação de strings de caracteres iniciada no Capítulo 8.

A biblioteca de tratamento de caracteres inclui várias funções que realizam testes úteis e manipulações de dados de caracteres. Cada função recebe um caractere — representado como um `int` — ou EOF como um argumento. Os caracteres são frequentemente manipulados como inteiros. Lembre-se de que EOF normalmente tem o valor `-1` e que algumas arquiteturas de hardware não permitem que valores negativos sejam armazenados em variáveis `char`. Portanto, as funções tratamento de caracteres manipulam os caracteres como inteiros. A Figura 22.17 resume as funções da biblioteca de tratamento de caracteres. Ao utilizar funções da biblioteca de tratamento de caracteres, inclua o arquivo de cabeçalho `<cctype>`.

Protótipo	Descrição
<code>int isdigit( int c )</code>	Retorna <code>true</code> se <code>c</code> for um dígito e <code>false</code> , caso contrário.
<code>int isalpha( int c )</code>	Retorna <code>true</code> se <code>c</code> for uma letra e <code>false</code> , caso contrário.
<code>int isalnum( int c )</code>	Retorna <code>true</code> se <code>c</code> for um dígito ou uma letra e <code>false</code> , caso contrário.
<code>int isxdigit( int c )</code>	Retorna <code>true</code> se <code>c</code> for caractere de dígito hexadecimal e <code>false</code> , caso contrário. (Ver o Apêndice D, “Sistemas de numeração”, para obter uma explicação detalhada dos números binário, octal, decimal e hexadecimal.)
<code>int islower( int c )</code>	Retorna <code>true</code> se <code>c</code> for uma letra minúscula e <code>false</code> , caso contrário.
<code>int isupper( int c )</code>	Retorna <code>true</code> se <code>c</code> for uma letra maiúscula e <code>false</code> , caso contrário.
<code>int tolower( int c )</code>	Se <code>c</code> for uma letra maiúscula, <code>tolower</code> retorna <code>c</code> como uma letra minúscula. Caso contrário, <code>tolower</code> retorna o argumento inalterado.
<code>int toupper( int c )</code>	Se <code>c</code> for uma letra minúscula, <code>toupper</code> retorna <code>c</code> como uma letra maiúscula. Caso contrário, <code>toupper</code> retorna o argumento inalterado.
<code>int isspace( int c )</code>	Retorna <code>true</code> se <code>c</code> for um caractere de espaço em branco — nova linha ( <code>'\n'</code> ), espaço ( <code>' '</code> ), avanço de formulário ( <code>'\f'</code> ), retorno de carro ( <code>'\r'</code> ), tabulação horizontal ( <code>'\t'</code> ) ou vertical ( <code>'\v'</code> ) — e <code>false</code> , caso contrário.
<code>int iscntrl( int c )</code>	Retorna <code>true</code> se <code>c</code> for um caractere de controle, como nova linha ( <code>'\n'</code> ), avanço de formulário ( <code>'\f'</code> ), retorno de carro ( <code>'\r'</code> ), tabulação horizontal ( <code>'\t'</code> ), tabulação vertical ( <code>'\v'</code> ), alerta ( <code>'\a'</code> ) ou backspace ( <code>'\b'</code> ) — e <code>false</code> , caso contrário.
<code>int ispunct( int c )</code>	Retorna <code>true</code> se <code>c</code> for um caractere de impressão diferente de um espaço, um dígito ou uma letra, e <code>false</code> , caso contrário.
<code>int isprint( int c )</code>	Retorna valor <code>true</code> se <code>c</code> for um caractere de impressão incluindo espaço ( <code>' '</code> ), e <code>false</code> , caso contrário.
<code>int isgraph( int c )</code>	Retorna <code>true</code> se <code>c</code> for um caractere de impressão diferente do espaço ( <code>' '</code> ), e <code>false</code> , caso contrário.

**Figura 22.17** Funções da biblioteca de tratamento de caracteres.

A Figura 22.18 demonstra as funções `isdigit`, `isalpha`, `isalnum` e `isxdigit`. A função `isdigit` determina se seu argumento é um dígito (0–9). A função `isalpha` determina se seu argumento é uma letra maiúscula (A–Z) ou uma letra minúscula (a–z). A função `isalnum` determina se seu argumento é uma letra maiúscula, uma minúscula ou um dígito. A função `isxdigit` determina se seu argumento é um dígito hexadecimal (A–F, a–f, 0–9).

A Figura 22.18 utiliza o operador condicional (`?:`) em cada função para determinar se a string `" is a "` ou `" is not a "` deve ser impressa na saída de cada caractere testado. Por exemplo, a linha 16 indica que se `'8'` for um dígito — isto é, se `isdigit` retorna um valor verdadeiro (não-zero) — a string `"8 is a "` é impressa. Se `'8'` não for um dígito (isto é, se `isdigit` retornar 0), a string `"8 is not a "` é impressa.

```

1 // Figura 22.18: fig22_18.cpp
2 // Utilizando as funções isdigit, isalpha, isalnum e isxdigit.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cctype> // protótipos de função de tratamento de caracteres
8 using std::isalnum;
9 using std::isalpha;
10 using std::isdigit;
11 using std::isxdigit;
12
```

**Figura 22.18** Funções de tratamento de caracteres `isdigit`, `isalpha`, `isalnum` e `isxdigit`.

(continua)

```

13 int main()
14 {
15 cout << "According to isdigit:\n"
16 << (isdigit('8') ? "8 is a" : "8 is not a") << " digit\n"
17 << (isdigit('#') ? "# is a" : "# is not a") << " digit\n";
18
19 cout << "\nAccording to isalpha:\n"
20 << (isalpha('A') ? "A is a" : "A is not a") << " letter\n"
21 << (isalpha('b') ? "b is a" : "b is not a") << " letter\n"
22 << (isalpha('&') ? "& is a" : "& is not a") << " letter\n"
23 << (isalpha('4') ? "4 is a" : "4 is not a") << " letter\n";
24
25 cout << "\nAccording to isalnum:\n"
26 << (isalnum('A') ? "A is a" : "A is not a")
27 << " digit or a letter\n"
28 << (isalnum('8') ? "8 is a" : "8 is not a")
29 << " digit or a letter\n"
30 << (isalnum('#') ? "# is a" : "# is not a")
31 << " digit or a letter\n";
32
33 cout << "\nAccording to isxdigit:\n"
34 << (isxdigit('F') ? "F is a" : "F is not a")
35 << " hexadecimal digit\n"
36 << (isxdigit('J') ? "J is a" : "J is not a")
37 << " hexadecimal digit\n"
38 << (isxdigit('7') ? "7 is a" : "7 is not a")
39 << " hexadecimal digit\n"
40 << (isxdigit('$') ? "$ is a" : "$ is not a")
41 << " hexadecimal digit\n"
42 << (isxdigit('f') ? "f is a" : "f is not a")
43 << " hexadecimal digit" << endl;
44 return 0;
45 } // fim do main

```

According to isdigit:

8 is a digit  
# is not a digit

According to isalpha:

A is a letter  
b is a letter  
& is not a letter  
4 is not a letter

According to isalnum:

A is a digit or a letter  
8 is a digit or a letter  
# is not a digit or a letter

According to isxdigit:

F is a hexadecimal digit  
J is not a hexadecimal digit  
7 is a hexadecimal digit  
\$ is not a hexadecimal digit  
f is a hexadecimal digit

**Figura 22.18** Funções de tratamento de caracteres isdigit, isalpha, isalnum e isxdigit.

(continuação)

A Figura 22.19 demonstra as funções `islower`, `isupper`, `tolower` e `toupper`. A função `islower` determina se seu argumento é uma letra minúscula (a–z). A função `isupper` determina se seu argumento é uma letra maiúscula (A–Z). A função `tolower` converte uma letra maiúscula em minúscula e retorna a minúscula — se o argumento não for uma letra maiúscula, `tolower` retorna o valor de argumento inalterado. A função `toupper` converte uma letra minúscula em maiúscula e retorna a letra maiúscula — se o argumento não for uma letra minúscula, `toupper` retorna o valor de argumento inalterado.

A Figura 22.20 demonstra as funções `isspace`, `iscntrl`, `ispunct`, `isprint` e `isgraph`. A função `isspace` determina se seu argumento é um caractere de espaço em branco, como espaço (' '), avanço de formulário ('\f'), nova linha ('\n'), retorno de carro ('\r'), tabulação horizontal ('\t') ou vertical ('\v'). A função `iscntrl` determina se seu argumento é um caractere de controle como tabulação horizontal ('\t'), tabulação vertical ('\v'), avanço de formulário ('\f'), alerta ('\a'), backspace ('\b'), retorno de carro ('\r') ou nova linha ('\n'). A função `ispunct` determina se seu argumento é um caractere de impressão diferente de um espaço, dígito

```

1 // Figura 22.19: fig22_19.cpp
2 // Utilizando as funções islower, isupper, tolower e toupper.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cctype> // protótipos de função de tratamento de caracteres
8 using std::islower;
9 using std::isupper;
10 using std::tolower;
11 using std::toupper;
12
13 int main()
14 {
15 cout << "According to islower:\n"
16 << (islower('p') ? "p is a" : "p is not a")
17 << " lowercase letter\n"
18 << (islower('P') ? "P is a" : "P is not a")
19 << " lowercase letter\n"
20 << (islower('5') ? "5 is a" : "5 is not a")
21 << " lowercase letter\n"
22 << (islower('!') ? "! is a" : "! is not a")
23 << " lowercase letter\n";
24
25 cout << "\nAccording to isupper:\n"
26 << (isupper('D') ? "D is an" : "D is not an")
27 << " uppercase letter\n"
28 << (isupper('d') ? "d is an" : "d is not an")
29 << " uppercase letter\n"
30 << (isupper('8') ? "8 is an" : "8 is not an")
31 << " uppercase letter\n"
32 << (isupper('$') ? "$ is an" : "$ is not an")
33 << " uppercase letter\n";
34
35 cout << "\nu converted to uppercase is "
36 << static_cast< char >(toupper('u'))
37 << "\n7 converted to uppercase is "
38 << static_cast< char >(toupper('7'))
39 << "\n$ converted to uppercase is "
40 << static_cast< char >(toupper('$'))
41 << "\nL converted to lowercase is "
42 << static_cast< char >(tolower('L')) << endl;
43 return 0;
44 } // fim do main

```

**Figura 22.19** Funções de tratamento de caracteres `islower`, `isupper`, `tolower` e `toupper`.

(continua)

```

According to islower:
p is a lowercase letter
P is not a lowercase letter
5 is not a lowercase letter
! is not a lowercase letter

According to isupper:
D is an uppercase letter
d is not an uppercase letter
8 is not an uppercase letter
$ is not an uppercase letter

u converted to uppercase is U
7 converted to uppercase is 7
$ converted to uppercase is $
L converted to lowercase is l

```

**Figura 22.19** Funções de tratamento de caracteres `islower`, `isupper`, `tolower` e `toupper`.

(continuação)

ou letra, como \$, #, (, ), [, ], {, }, :, ou %. A função `isprint` determina se seu argumento é um caractere que pode ser exibido na tela (incluindo o caractere de espaço em branco). A função `isgraph` testa os mesmos caracteres que `isprint`; entretanto, o caractere de espaço em branco não é incluído.

```

1 // Figura 22.20: fig22_20.cpp
2 // Utilizando as funções isspace, iscntrl, ispunct, isprint, isgraph.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cctype> // protótipos de função de tratamento de caracteres
8 using std::iscntrl;
9 using std::isgraph;
10 using std::isprint;
11 using std::ispunct;
12 using std::isspace;
13
14 int main()
15 {
16 cout << "According to isspace:\nNewline "
17 << (isspace('\n') ? "is a" : "is not a")
18 << " whitespace character\nHorizontal tab "
19 << (isspace('\t') ? "is a" : "is not a")
20 << " whitespace character\n"
21 << (isspace('%') ? "% is a" : "% is not a")
22 << " whitespace character\n";
23
24 cout << "\nAccording to iscntrl:\nNewline "
25 << (iscntrl('\n') ? "is a" : "is not a")
26 << " control character\n"
27 << (iscntrl('$') ? "$ is a" : "$ is not a")
28 << " control character\n";
29

```

**Figura 22.20** Funções de tratamento de caracteres `isspace`, `iscntrl`, `ispunct`, `isprint` e `isgraph`.

(continua)

```

30 cout << "\nAccording to ispunct:\n"
31 << (ispunct(';') ? ";" is a" : ";" is not a")
32 << " punctuation character\n"
33 << (ispunct('Y') ? "Y is a" : "Y is not a")
34 << " punctuation character\n"
35 << (ispunct('#') ? "# is a" : "# is not a")
36 << " punctuation character\n";
37
38 cout << "\nAccording to isprint:\n"
39 << (isprint('$') ? "$ is a" : "$ is not a")
40 << " printing character\nAlert "
41 << (isprint('\a') ? "is a" : "is not a")
42 << " printing character\nSpace "
43 << (isprint(' ') ? "is a" : "is not a")
44 << " printing character\n";
45
46 cout << "\nAccording to isgraph:\n"
47 << (isgraph('Q') ? "Q is a" : "Q is not a")
48 << " printing character other than a space\nSpace "
49 << (isgraph(' ') ? "is a" : "is not a")
50 << " printing character other than a space" << endl;
51 return 0;
52 } // fim do main

```

According to isspace:  
 Newline is a whitespace character  
 Horizontal tab is a whitespace character  
 % is not a whitespace character

According to iscntrl:  
 Newline is a control character  
 \$ is not a control character

According to ispunct:  
 ; is a punctuation character  
 Y is not a punctuation character  
 # is a punctuation character

According to isprint:  
 \$ is a printing character  
 Alert is not a printing character  
 Space is a printing character

According to isgraph:  
 Q is a printing character other than a space  
 Space is not a printing character other than a space

**Figura 22.20** Funções de tratamento de caracteres isspace, iscntrl, ispunct, isprint e isgraph.

(continuação)

## 22.10 Funções de conversão de string baseada em ponteiro

No Capítulo 8, discutimos várias das funções de manipulação de string baseadas em ponteiro do C++ mais populares. Nas próximas seções, abordaremos as funções restantes, incluindo funções para converter strings em valores numéricos, funções para pesquisar strings e funções para manipular, comparar e pesquisar blocos de memória.

Esta seção apresenta as **funções de conversão de strings** baseadas em ponteiro da **biblioteca de utilitários gerais <cstdlib>**. Essas funções convertem strings de caracteres baseadas em ponteiros em valores do tipo inteiro e ponto flutuante. A Figura 22.21 resume as funções de conversão de strings baseadas em ponteiros. Observe o uso de `const` para declarar a variável `nPtr` nos cabeçalhos de

Protótipo	Descrição
<code>double atof( const char *nPtr )</code>	Converte a string nPtr em double. Se a string não pode ser convertida, 0 é retornado.
<code>int atoi( const char *nPtr )</code>	Converte a string nPtr em int. Se a string não pode ser convertida, 0 é retornado.
<code>long atol( const char *nPtr )</code>	Converte a string nPtr em long int. Se a string não pode ser convertida, 0 é retornado.
<code>double strtod( const char *nPtr, char **endPtr )</code>	Converte a string nPtr em double. endPtr é o endereço de um ponteiro para o resto da string depois do double. Se a string não pode ser convertida, 0 é retornado.
<code>long strtol( const char *nPtr, char **endPtr, int base )</code>	Converte a string nPtr em long. endPtr é o endereço de um ponteiro para o resto da string depois do long. Se a string não pode ser convertida, 0 é retornado. O parâmetro base indica a base do número para converter (por exemplo, 8 para octal, 10 para decimal ou 16 para hexadecimal). O padrão é decimal.
<code>unsigned long strtoul( const char *nPtr, char **endPtr, int base )</code>	Converte a string nPtr em unsigned long. endPtr é o endereço de um ponteiro para o resto da string depois do unsigned long. Se a string não pode ser convertida, 0 é retornado. O parâmetro base indica a base do número para converter (por exemplo, 8 para octal, 10 para decimal ou 16 para hexadecimal). O padrão é decimal.

**Figura 22.21** As funções de conversão de string baseadas em ponteiro da biblioteca de utilitários gerais.

função (lidos da direita para a esquerda como ‘nPtr é um ponteiro para um constante de caractere’). Ao utilizar funções da biblioteca de utilitários gerais, inclua o arquivo de cabeçalho `<cstdlib>`.

A função **atof** (Figura 22.22, linha 12) converte seu argumento — uma string que representa um número de ponto flutuante — em um valor `double`. A função retorna o valor `double`. Se a string não pode ser convertida — por exemplo, se o primeiro caractere da string não for um dígito — a função **atof** retorna zero.

A função **atoi** (Figura 22.23, linha 12) converte seu argumento — uma string de dígitos que representa um inteiro — em um valor `int`. A função retorna o valor `int`. Se a string não pode ser convertida, a função **atoi** retorna zero.

```

1 // Figura 22.22: fig22_22.cpp
2 // Utilizando atof.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstdlib> // protótipo de atof
8 using std::atof;
9
10 int main()
11 {
12 double d = atof("99.0"); // converte string em double
13
14 cout << "The string \"99.0\" converted to double is " << d
15 << "\nThe converted value divided by 2 is " << d / 2.0 << endl;
16 return 0;
17 } // fim do main

```

```

The string "99.0" converted to double is 99
The converted value divided by 2 is 49.5

```

**Figura 22.22** Função de conversão de strings **atof**.



```

1 // Figura 22.23: Fig22_23.cpp
2 // Utilizando atoi.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstdlib> // protótipo de atoi
8 using std::atoi;
9
10 int main()
11 {
12 int i = atoi("2593"); // converte string em int
13
14 cout << "The string \"2593\" converted to int is " << i
15 << "\nThe converted value minus 593 is " << i - 593 << endl;
16 return 0;
17 } // fim do main

```

```

The string "2593" converted to int is 2593
The converted value minus 593 is 2000

```

**Figura 22.23** Função de conversão de strings `atoi`.

A função `atoi` (Figura 22.24, linha 12) converte seu argumento — uma string de dígitos que representa um inteiro do tipo `long` — em um valor `long`. A função retorna o valor `long`. Se a string não pode ser convertida, a função `atoi` retorna zero. Se `int` e `long` são ambos armazenados em quatro bytes, a função `atoi` e a função `atol` funcionam de modo idêntico.

A função `strtod` (Figura 22.25) converte uma sequência de caracteres que representa um valor de ponto flutuante em `double`. A função `strtod` recebe dois argumentos — uma string (`char *`) e o endereço de um ponteiro `char *` (isto é, um `char **`). A string contém a sequência de caracteres a ser convertida em `double`. O segundo argumento permite que `strtod` modifique um ponteiro `char *` na função chamadora, de tal modo que o ponteiro aponte para a localização do primeiro caractere depois da parte convertida da string.

```

1 // Figura 22.24: fig22_24.cpp
2 // Utilizando atol.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstdlib> // protótipo de atol
8 using std::atol;
9
10 int main()
11 {
12 long x = atol("1000000"); // converte string em long
13
14 cout << "The string \"1000000\" converted to long is " << x
15 << "\nThe converted value divided by 2 is " << x / 2 << endl;
16 return 0;
17 } // fim do main

```

```

The string "1000000" converted to long int is 1000000
The converted value divided by 2 is 500000

```

**Figura 22.24** Função de conversão de string `atol`.

```

1 // Figura 22.25: fig22_25.cpp
2 // Utilizando strtod.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstdlib> // protótipo de strtod
8 using std::strtod;
9
10 int main()
11 {
12 double d;
13 const char *string1 = "51.2% are admitted";
14 char *stringPtr;
15
16 d = strtod(string1, &stringPtr); // converte caracteres em double
17
18 cout << "The string \"" << string1
19 << "\" is converted to the\ndouble value " << d
20 << " and the string \"" << stringPtr << "\" << endl;
21 return 0;
22 } // fim do main

```

The string "51.2% are admitted" is converted to the double value 51.2 and the string "% are admitted"

**Figura 22.25** Função de conversão de strings `strtod`.

A linha 16 indica que o valor `double` convertido a partir de `string` é atribuído a `d` e que a localização do primeiro caractere depois do valor convertido (51.2) em `string` é atribuída a `stringPtr`.

A função `strtol` (Figura 22.26) converte em `long` uma sequência de caracteres que representam um inteiro. A função recebe três argumentos — uma `string` (`char *`), o endereço de um ponteiro `char *` e um inteiro. A `string` contém a sequência de caracteres a converter. A localização do primeiro caractere depois da parte convertida da `string` é atribuída ao segundo argumento. O inteiro especifica a *base* do valor sendo convertido. A linha 16 indica que o valor `long` convertido a partir de `string1` é atribuído a `x` e que a localização do primeiro caractere depois do valor convertido (-1234567) em `string` é atribuída a `remainderPtr`. Utilizar um ponteiro nulo para o segundo argumento faz com que o restante da `string` seja ignorado. O terceiro argumento, 0, indica que o valor a ser convertido pode estar em octal (base 8), decimal (base 10) ou hexadecimal (base 16). Isso é determinado pelos caracteres iniciais na `string` — 0 indica um número octal, 0x indica hexadecimal e um número de 1–9 indica decimal.

Em uma chamada à função `strtol`, a base pode ser especificada como zero ou como qualquer valor entre 2 e 36. (Ver o Apêndice D para uma explicação detalhada dos sistemas de número octal, decimal, hexadecimal e binário.) As representações numéricas de inteiros desde a base 11 até a base 36 utilizam os caracteres A–Z para representar os valores 10 a 35. Por exemplo, os valores hexadecimais podem consistir nos dígitos 0–9 e nos caracteres A–F. Um inteiro de base 11 pode consistir nos dígitos 0–9 e no caractere A. Um inteiro de base 24 pode consistir nos dígitos 0–9 e nos caracteres A–N. Um inteiro de base 36 pode consistir nos dígitos 0–9 e nos caracteres A–Z. [Nota: A caixa (caixa alta ou baixa) da letra utilizada é ignorada.]

A função `strtoul` (Figura 22.27) converte em `unsigned long` uma sequência de caracteres que representa um inteiro `unsigned long`. A função funciona identicamente à função `strtol`. A linha 17 indica que o valor `unsigned long` convertido de `string` é atribuído ao `x` e que a localização do primeiro caractere depois do valor convertido (1234567) em `string1` é atribuída a `remainderPtr`. O terceiro argumento, 0, indica que o valor a ser convertido pode estar no formato octal, decimal ou hexadecimal, dependendo dos caracteres iniciais.

```

1 // Figura 22.26: fig22_26.cpp
2 // Utilizando strtol.
3 #include <iostream>

```

**Figura 22.26** Função de conversão de strings `strtol`.

(continua)

```

4 using std::cout;
5 using std::endl;
6
7 #include <cstdlib> // protótipo de strtol
8 using std::strtol;
9
10 int main()
11 {
12 long x;
13 const char *string1 = "-1234567abc";
14 char *remainderPtr;
15
16 x = strtol(string1, &remainderPtr, 0); // converte caracteres em long
17
18 cout << "The original string is \"" << string1
19 << "\"\n" << "The converted value is " << x
20 << "\n" << "The remainder of the original string is \"" << remainderPtr
21 << "\"\n" << "The converted value plus 567 is " << x + 567 << endl;
22 return 0;
23 } // fim do main

```

The original string is "-1234567abc"  
The converted value is -1234567  
The remainder of the original string is "abc"  
The converted value plus 567 is -1234000

**Figura 22.26** Função de conversão de strings `strtol`.

(continuação)

```

1 // Figura 22.27: fig22_27.cpp
2 // Utilizando strtoul.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstdlib> // protótipo de strtoul
8 using std::strtoul;
9
10 int main()
11 {
12 unsigned long x;
13 const char *string1 = "1234567abc";
14 char *remainderPtr;
15
16 // converte uma sequência de caracteres em unsigned long
17 x = strtoul(string1, &remainderPtr, 0);
18
19 cout << "The original string is \"" << string1
20 << "\"\n" << "The converted value is " << x
21 << "\n" << "The remainder of the original string is \"" << remainderPtr
22 << "\"\n" << "The converted value minus 567 is " << x - 567 << endl;
23 return 0;
24 } // fim do main

```

**Figura 22.27** Função de conversão de strings `strtoul`.

(continua)

```
The original string is "1234567abc"
The converted value is 1234567
The remainder of the original string is "abc"
The converted value minus 567 is 1234000
```

**Figura 22.27** Função de conversão de strings `strtoul`.

(continuação)

## 22.11 Funções de pesquisa da biblioteca de tratamento de strings baseadas em ponteiro

Esta seção apresenta as funções da biblioteca de tratamento utilizadas para pesquisar caracteres em strings e outras strings. As funções são resumidas na Figura 22.28. Observe que as funções `strcspn` e `strspn` especificam o tipo de retorno `size_t`. O tipo `size_t` é um tipo definido pelo padrão como o tipo inteiro do valor retornado pelo operador `sizeof`.



### Dica de portabilidade 22.6

*O tipo `size_t` é um sinônimo dependente de sistema para o tipo `unsigned long` ou o tipo `unsigned int`.*

A função `strchr` procura a primeira ocorrência de um caractere em uma string. Se o caractere é localizado, `strchr` retorna um ponteiro para o caractere na string; caso contrário, `strchr` retorna um ponteiro nulo. O programa da Figura 22.29 utiliza `strchr` (linhas 17 e 25) para procurar as primeiras ocorrências de 'a' e 'z' na string "This is a test".

A função `strcspn` (Figura 22.30, linha 18) determina o comprimento da parte inicial da string em seu primeiro argumento que não contém caracteres da string em seu segundo argumento. A função retorna o comprimento do segmento.

A função `strpbrk` procura a primeira ocorrência em seu primeiro argumento de string de qualquer caractere em seu segundo argumento de string. Se um caractere do segundo argumento é localizado, `strpbrk` retorna um ponteiro para o caractere no primeiro argumento; caso contrário, `strpbrk` retorna um ponteiro nulo. A linha 16 da Figura 22.31 localiza a primeira ocorrência na string1 de qualquer caractere a partir da string2.

A função `strrchr` procura a última ocorrência do caractere especificado em uma string. Se o caractere é localizado, `strrchr` retorna um ponteiro para o caractere na string; caso contrário, `strrchr` retorna 0. A linha 18 da Figura 22.32 procura a última ocorrência do caractere 'z' na string "A zoo has many animals including zebras".

Protótipo	Descrição
<code>char *strchr( const char *s, int c )</code>	Localiza a primeira ocorrência do caractere <code>c</code> na string <code>s</code> . Se <code>c</code> for localizado, um ponteiro para <code>c</code> em <code>s</code> é retornado. Caso contrário, um ponteiro nulo é retornado.
<code>char *strrchr( const char *s, int c )</code>	Pesquisa a partir do final da string <code>s</code> e localiza a última ocorrência do caractere <code>c</code> na string <code>s</code> . Se <code>c</code> é localizado, um ponteiro para <code>c</code> na string <code>s</code> é retornado. Caso contrário, um ponteiro nulo é retornado.
<code>size_t strspn( const char *s1, const char *s2 )</code>	Determina e retorna o comprimento do segmento inicial da string <code>s1</code> que consiste somente nos caracteres contidos na string <code>s2</code> .
<code>char *strpbrk( const char *s1, const char *s2 )</code>	Localiza a primeira ocorrência na string <code>s1</code> de qualquer caractere na string <code>s2</code> . Se um caractere de string <code>s2</code> é localizado, um ponteiro para o caractere na string <code>s1</code> é retornado. Caso contrário, um ponteiro nulo é retornado.
<code>size_t strcspn( const char *s1, const char *s2 )</code>	Determina e retorna o comprimento do segmento inicial da string <code>s1</code> que consiste nos caracteres não contidos na string <code>s2</code> .
<code>char *strstr( const char *s1, const char *s2 )</code>	Localiza a primeira ocorrência na string <code>s1</code> da string <code>s2</code> . Se a string é localizada, um ponteiro para a string em <code>s1</code> é retornado. Caso contrário, um ponteiro nulo é retornado.

**Figura 22.28** Funções de pesquisa da biblioteca de tratamento de strings baseadas em ponteiro.

```

1 // Figura 22.29: fig22_29.cpp
2 // Utilizando strchr.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // protótipo de strchr
8 using std::strchr;
9
10 int main()
11 {
12 const char *string1 = "This is a test";
13 char character1 = 'a';
14 char character2 = 'z';
15
16 // procura character1 em string1
17 if (strchr(string1, character1) != NULL)
18 cout << '\'' << character1 << "' was found in \""
19 << string1 << "\".\n";
20 else
21 cout << '\'' << character1 << "' was not found in \""
22 << string1 << "\".\n";
23
24 // procura character2 em string1
25 if (strchr(string1, character2) != NULL)
26 cout << '\'' << character2 << "' was found in \""
27 << string1 << "\".\n";
28 else
29 cout << '\'' << character2 << "' was not found in \""
30 << string1 << "\"." << endl;
31
32 return 0;
33 } // fim do main

```

```

'a' was found in "This is a test".
'z' was not found in "This is a test".

```

**Figura 22.29** Função de pesquisa de string strchr.

A função **strspn** (Figura 22.33, linha 18) determina o comprimento da parte inicial da string em seu primeiro argumento que contém somente caracteres da string em seu segundo argumento. A função retorna o comprimento do segmento.

A função **strstr** procura a primeira ocorrência de seu segundo argumento de string em seu primeiro argumento de string. Se a segunda string é localizada na primeira string, um ponteiro para a localização da string no primeiro argumento é retornado; caso contrário, retorna 0. A linha 18 da Figura 22.34 utiliza **strstr** para localizar a string "def" na string "abcdefabcdef".

## 22.12 Funções de memória da biblioteca de tratamento de strings baseadas em ponteiro

As funções da biblioteca de tratamento de strings apresentadas nesta seção facilitam a manipulação, comparação e pesquisa de blocos de memória. As funções tratam os blocos de memória como arrays de bytes. Essas funções podem manipular qualquer bloco de dados. A Figura 22.35 resume as funções de memória da biblioteca de tratamento de strings. Nas discussões da função, ‘objeto’ refere-se a um bloco de dados. [Nota: As funções de processamento de string nas seções anteriores operam em strings de caractere terminadas por caractere nulo. As funções nesta seção operam em arrays de bytes. O valor de caractere nulo (isto é, um byte contendo 0) não tem nenhuma importância com as funções nesta seção.]

```

1 // Figura 22.30: fig22_30.cpp
2 // Utilizando strchr.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // protótipo de strchr
8 using std::strchr;
9
10 int main()
11 {
12 const char *string1 = "The value is 3.14159";
13 const char *string2 = "1234567890";
14
15 cout << "string1 = " << string1 << "\nstring2 = " << string2
16 << "\n\nThe length of the initial segment of string1"
17 << "\ncontaining no characters from string2 = "
18 << strchr(string1, string2) << endl;
19 return 0;
20 } // fim do main

```

string1 = The value is 3.14159  
string2 = 1234567890

The length of the initial segment of string1  
containing no characters from string2 = 13

**Figura 22.30** Função de pesquisa de string strchr.

```

1 // Figura 22.31: fig22_31.cpp
2 // Utilizando strpbrk.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // protótipo de strpbrk
8 using std::strpbrk;
9
10 int main()
11 {
12 const char *string1 = "This is a test";
13 const char *string2 = "beware";
14
15 cout << "Of the characters in \"" << string2 << "\"\n"
16 << *strpbrk(string1, string2) << "' is the first character "
17 << "to appear in\n\"" << string1 << "\"\n" << endl;
18 return 0;
19 } // fim do main

```

Of the characters in "beware"  
'a' is the first character to appear in  
"This is a test"

**Figura 22.31** Função de pesquisa de string strpbrk.

```

1 // Figura 22.32: fig22_32.cpp
2 // Utilizando strchr.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // protótipo de strchr
8 using std::strchr;
9
10 int main()
11 {
12 const char *string1 = "A zoo has many animals including zebras";
13 char c = 'z';
14
15 cout << "string1 = " << string1 << "\n" << endl;
16 cout << "The remainder of string1 beginning with the\n"
17 << "last occurrence of character '"
18 << c << "' is: \"" << strchr(string1, c) << "'" << endl;
19 return 0;
20 } // fim do main

```

string1 = A zoo has many animals including zebras

The remainder of string1 beginning with the  
last occurrence of character 'z' is: "zebras"

**Figura 22.32** Função de pesquisa de string strchr.

```

1 // Figura 22.33: fig22_33.cpp
2 // Utilizando strspn.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // protótipo de strspn
8 using std::strspn;
9
10 int main()
11 {
12 const char *string1 = "The value is 3.14159";
13 const char *string2 = "aehils Tuv";
14
15 cout << "string1 = " << string1 << "\nstring2 = " << string2
16 << "\n\nThe length of the initial segment of string1\n"
17 << "containing only characters from string2 = "
18 << strspn(string1, string2) << endl;
19 return 0;
20 } // fim do main

```

string1 = The value is 3.14159  
string2 = aehils Tuv

The length of the initial segment of string1  
containing only characters from string2 = 13

**Figura 22.33** Função de pesquisa de string strspn.



```

1 // Figura 22.34: fig22_34.cpp
2 // Utilizando strstr.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // protótipo de strstr
8 using std::strstr;
9
10 int main()
11 {
12 const char *string1 = "abcdefabcdef";
13 const char *string2 = "def";
14
15 cout << "string1 = " << string1 << "\nstring2 = " << string2
16 << "\n\nThe remainder of string1 beginning with the\n"
17 << "first occurrence of string2 is: "
18 << strstr(string1, string2) << endl;
19 return 0;
20 } // fim do main

```

```

string1 = abcdefabcdef
string2 = def

```

```

The remainder of string1 beginning with the
first occurrence of string2 is: defabcdef

```

**Figura 22.34** Função de pesquisa de string strstr.

Protótipo	Descrição
<code>void *memcpy( void *s1, const void *s2, size_t n )</code>	Copia os caracteres n do objeto apontado por s2 no objeto apontado por s1. Um ponteiro para o objeto resultante é retornado. A área a partir da qual os caracteres são copiados não tem permissão de se sobrepor à área em que os caracteres são copiados.
<code>void *memmove( void *s1, const void *s2, size_t n )</code>	Copia os caracteres n do objeto apontado por s2 no objeto apontado por s1. A cópia é realizada como se os caracteres fossem os primeiros copiados do objeto apontado por s2 em um array temporário e, então, copiados do array temporário no objeto apontado por s1. Um ponteiro para o objeto resultante é retornado. A área a partir da qual os caracteres são copiados tem permissão de se sobrepor à área em que os caracteres são copiados.
<code>int memcmp( const void *s1, const void *s2, size_t n )</code>	Compara os primeiros caracteres n dos objetos apontados por s1 e s2. A função retorna 0, menor que 0 ou maior que 0 se s1 for igual a, menor que ou maior que s2, respectivamente.
<code>void *memchr( const void *s, int c, size_t n )</code>	Localiza a primeira ocorrência de c (convertido em unsigned char) nos primeiros caracteres n do objeto apontado por s. Se c é localizado, um ponteiro para c no objeto é retornado. Caso contrário, 0 é retornado.
<code>void *memset( void *s, int c, size_t n )</code>	Copia c (convertido em unsigned char) nos primeiros caracteres n do objeto apontado por s. Um ponteiro para o resultado é retornado.

**Figura 22.35** Funções de memória da biblioteca de tratamento de strings.

Os parâmetros de ponteiro para essas funções são declarados `void *`. No Capítulo 8, vimos que um ponteiro para qualquer tipo de dados pode ser atribuído diretamente a um ponteiro do tipo `void *`. Por essa razão, essas funções podem receber ponteiros de qualquer tipo de dados. Lembre-se de que um ponteiro do tipo `void *` não pode ser atribuído diretamente a um ponteiro de qualquer outro tipo de dados. Como um ponteiro `void *` não pode ser desreferenciado, cada função recebe um argumento de tamanho que especifica o número de caracteres (bytes) que a função processará. Para simplificar, os exemplos desta seção manipulam arrays de caractere (blocos de caracteres).

A função **memcpy** copia um número especificado de caracteres (bytes) do objeto apontado por seu segundo argumento no objeto apontado por seu primeiro argumento. A função pode receber um ponteiro para qualquer tipo de objeto. O resultado dessa função é indefinido se os dois objetos se sobrepõem na memória (isto é, são partes do mesmo objeto). O programa da Figura 22.36 utiliza **memcpy** (linha 17) para copiar a string do array `s2` para o array `s1`.

A função **memmove**, como a **memcpy**, copia um número especificado de bytes do objeto apontado por seu segundo argumento no objeto apontado por seu primeiro argumento. A cópia é realizada como se os bytes fossem copiados do segundo argumento para um array temporário de caracteres e, então, copiados do array temporário para o primeiro argumento. Isso permite aos caracteres de uma parte de uma string ser copiados em outra parte da mesma string.



## Erro comum de programação 22.8

*As funções de manipulação de string diferentes de **memmove** que copiam caracteres têm resultados indefinidos quando a cópia acontece entre partes da mesma string.*

O programa da Figura 22.37 utiliza **memmove** (linha 16) para copiar os últimos 10 bytes de array `x` nos primeiros 10 bytes do array `x`.

A função **memcmp** (Figura 22.38, linhas 19, 20 e 21) compara o número especificado de caracteres de seu primeiro argumento com os caracteres correspondentes de seu segundo argumento. A função retorna um valor maior que zero se o primeiro argumento for maior que o segundo argumento, zero se os argumentos forem iguais e um valor menor que zero se o primeiro argumento for menor que o segundo argumento. [Nota: Com alguns compiladores, a função **memcmp** retorna -1, 0 ou 1, como na saída de exemplo da Figura 22.38. Com outros compiladores, essa função retorna 0 ou a diferença entre os códigos numéricos dos primeiros caracteres que diferem nas strings sendo comparadas. Por exemplo, quando `s1` e `s2` são comparados, o primeiro caractere que difere entre eles é o quinto caractere de cada string — E (código numérico 69) para `s1` e X (código numérico 72) para `s2`. Nesse caso, o valor de retorno será 19 (ou -19 quando `s2` for comparada com `s1`).]

```

1 // Figura 22.36: fig22_36.cpp
2 // Utilizando memcpy.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // protótipo de memcpy
8 using std::memcpy;
9
10 int main()
11 {
12 char s1[17];
13
14 // 17 caracteres totais (inclui caractere de terminação nulo)
15 char s2[] = "Copy this string";
16
17 memcpy(s1, s2, 17); // copia 17 caracteres de s2 para s1
18
19 cout << "After s2 is copied into s1 with memcpy,\n"
20 << "s1 contains \"" << s1 << "\"\n" << endl;
21 return 0;
22 } // fim do main

```

After s2 is copied into s1 with memcpy,  
s1 contains "Copy this string"

**Figura 22.36** Função de tratamento de memória **memcpy**.

A função `memchr` procura a primeira ocorrência de um byte, representado como `unsigned char`, no número especificado de bytes de um objeto. Se o byte é localizado no objeto, um ponteiro para ele é retornado; caso contrário, a função retorna um ponteiro nulo. A linha 16 da Figura 22.39 procura o caractere (byte) 'r' na string "This is a string".

```

1 // Figura 22.37: fig22_37.cpp
2 // Utilizando memmove.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // protótipo de memmove
8 using std::memmove;
9
10 int main()
11 {
12 char x[] = "Home Sweet Home";
13
14 cout << "The string in array x before memmove is: " << x;
15 cout << "\nThe string in array x after memmove is: "
16 << static_cast<char*>(memmove(x, &x[5], 10)) << endl;
17 return 0;
18 } // fim do main

```

The string in array x before memmove is: Home Sweet Home  
The string in array x after memmove is: Sweet Home Home

**Figura 22.37** Função de tratamento de memória `memmove`.

```

1 // Figura 22.38: fig22_38.cpp
2 // Utilizando memcmp.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::setw;
9
10 #include <cstring> // protótipo de memcmp
11 using std::memcmp;
12
13 int main()
14 {
15 char s1[] = "ABCDEFGH";
16 char s2[] = "ABCDXYZ";
17
18 cout << "s1 = " << s1 << "\ns2 = " << s2 << endl
19 << "\nmemcmp(s1, s2, 4) = " << setw(3) << memcmp(s1, s2, 4)
20 << "\nmemcmp(s1, s2, 7) = " << setw(3) << memcmp(s1, s2, 7)
21 << "\nmemcmp(s2, s1, 7) = " << setw(3) << memcmp(s2, s1, 7)
22 << endl;
23 return 0;
24 } // fim do main

```

**Figura 22.38** Função de tratamento de memória `memcmp`.

(continua)

```
s1 = ABCDEFG
s2 = ABCDXYZ

memcmp(s1, s2, 4) = 0
memcmp(s1, s2, 7) = -1
memcmp(s2, s1, 7) = 1
```

Figura 22.38 Função de tratamento de memória memcmp.

(continuação)

```
1 // Figura 22.39: fig22_39.cpp
2 // Utilizando memchr.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // protótipo de memchr
8 using std::memchr;
9
10 int main()
11 {
12 char s[] = "This is a string";
13
14 cout << "s = " << s << "\n" << endl;
15 cout << "The remainder of s after character 'r' is found is \""
16 << static_cast<char*>(memchr(s, 'r', 16)) << "\"" << endl;
17 return 0;
18 } // fim do main
```

```
s = This is a string
```

```
The remainder of s after character 'r' is found is "ring"
```

Figura 22.39 Função de tratamento de memória memchr.

A função `memset` copia o valor do byte em seu segundo argumento para um número especificado de bytes do objeto apontado por seu primeiro argumento. A linha 16 na Figura 22.40 utiliza `memset` para copiar 'b' nos primeiros 7 bytes de `string1`.

```
1 // Figura 22.40: fig22_40.cpp
2 // Utilizando memset.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // protótipo de memset
8 using std::memset;
9
10 int main()
11 {
12 char string1[15] = "BBBBBBBBBBBBBBB";
13 }
```

Figura 22.40 Função de tratamento de memória memset.

(continua)

```

14 cout << "string1 = " << string1 << endl;
15 cout << "string1 after memset = "
16 << static_cast<char*>(memset(string1, 'b', 7)) << endl;
17 return 0;
18 } // fim do main

```

```

string1 = BBBBBBBBBBBBBB
string1 after memset = bbbbbbbBBBBBBB

```

**Figura 22.40** Função de tratamento de memória `memset`.

(continuação)

## 22.13 Síntese

Este capítulo introduziu as definições `struct`, inicializando-as e utilizando-as com funções. Discutimos `typedef`, utilizando-o para criar aliases para ajudar a promover a portabilidade. Introduzimos também os operadores de bits para manipular dados e campos de bit para armazenar dados compactamente. Você também aprendeu sobre as funções de conversão de strings em `<cstring>` e as funções de processamento de string em `<string>`. No próximo capítulo, continuaremos nossa discussão de estruturas de dados discutindo contêineres — estruturas de dados definidas na C++ Standard Template Library. Apresentaremos também os muitos algoritmos definidos no STL.

### Resumo

- Estruturas são coleções de variáveis relacionadas (ou agregados) sob um nome.
- Elas podem conter variáveis de tipos de dados diferentes.
- A palavra-chave `struct` inicia cada definição de estrutura. Entre as chaves da definição de estrutura estão as declarações de membro de estrutura.
- Os membros da mesma estrutura devem ter nomes únicos.
- Uma definição de estrutura cria um novo tipo de dados que pode ser utilizado para declarar variáveis.
- Uma estrutura pode ser inicializada com uma lista inicializadora colocando-se depois da variável na declaração um sinal de igual e uma lista separada por vírgulas de inicializadores entre chaves. Se houver menos inicializadores na lista do que membros na estrutura, os membros restantes são inicializados como zero (ou um ponteiro nulo para membros de ponteiro).
- Variáveis de estrutura do mesmo tipo podem ser atribuídas a variáveis de estrutura inteiras.
- Uma variável de estrutura pode ser inicializada com uma variável de estrutura do mesmo tipo.
- As variáveis de estrutura e os membros de estrutura individuais são passados para funções por valor.
- Para passar uma estrutura por referência, passe o endereço da variável de estrutura ou uma referência à variável de estrutura. Um array de estruturas é passado por referência. Para passar um array por valor, crie uma estrutura com o array como um membro.
- Criar um novo nome de tipo com `typedef` não cria um novo tipo; cria um nome que é sinônimo de um tipo previamente definido.
- O operador E de bits (`&`) aceita dois operandos integrais. Um bit no resultado é configurado como um se os bits correspondentes em cada um dos operandos forem um.
- As máscaras são utilizadas com o E sobre bits para ocultar alguns bits enquanto preserva outros.
- O operador OU inclusivo sobre bits (`|`) aceita dois operandos. Um bit no resultado é configurado como um se o bit correspondente em qualquer operando for configurado como um.
- Cada um dos operadores de bits (exceto complemento) tem um operador de atribuição correspondente.
- O operador de OU exclusivo de bits (`^`) aceita dois operandos. Um bit no resultado é configurado como um se exatamente um dos bits correspondentes nos dois operandos for configurado como um.
- O operador de deslocamento para a esquerda (`<<`) desloca os bits de seu operando esquerdo para a esquerda pelo número de bits especificado por seu operando direito. Os bits tornados vagos à direita são substituídos por zeros.
- O operador de deslocamento para a direita (`>>`) desloca os bits de seu operando esquerdo para a direita pelo número de bits especificado em seu operando direito. Realizar um deslocamento para a direita em um inteiro sem sinal faz com que os bits tornados vagos à esquerda sejam substituídos por zeros. Bits vagos em inteiros com sinal podem ser substituídos por zeros ou uns — isso é dependente de máquina.
- O operador de complemento de bits (`~`) aceita um operando e inverte seus bits — isso produz o complemento de um do operando.

- Os campos de bit reduzem o uso de memória armazenando os dados no número mínimo de bits requerido. Os membros de campo de bit devem ser declarados como `int` ou `unsigned`.
- Um campo de bits é declarado colocando-se depois de um nome de membro `unsigned` ou `int` um caractere de dois-pontos e a largura do campo de bits.
- A largura de campo de bits deve ser de uma constante inteira.
- Se um campo de bit é especificado sem um nome, o campo é utilizado como preenchimento na estrutura.
- Um campo de bit não identificado com a largura 0 alinha o próximo campo de bit em um novo limite de palavra de máquina.
- A função `islower` determina se seu argumento é uma letra minúscula (a–z). A função `isupper` determina se seu argumento é uma letra maiúscula (A–Z).
- A função `isdigit` determina se seu argumento é um dígito (0–9).
- A função `isalpha` determina se seu argumento é uma letra maiúscula (A–Z) ou minúscula (a–z).
- A função `isalnum` determina se seu argumento é uma letra maiúscula (A–Z), uma minúscula (a–z) ou um dígito (0–9).
- A função `isxdigit` determina se seu argumento é um dígito hexadecimal (A–F, a–f, 0–9).
- A função `toupper` converte uma letra minúscula em maiúscula. A função `tolower` converte uma letra maiúscula em minúscula.
- A função `isspace` determina se seu argumento é um dos seguintes caracteres de espaço em branco: ' ' (espaço), '\f', '\n', '\r', '\t' ou '\v'.
- A função `isctrnl` determina se seu argumento é um caractere de controle, como '\t', '\v', '\f', '\a', '\b', '\r' ou '\n'.
- A função `ispunct` determina se seu argumento é um caractere de impressão diferente de um espaço, dígito ou letra.
- A função `isprint` determina se seu argumento é algum caractere de impressão, incluindo espaço.
- A função `isgraph` determina se seu argumento é um caractere de impressão diferente de espaço.
- A função `atof` converte seu argumento — uma string que inicia com uma série de dígitos que representa um número de ponto flutuante — para um valor `double`.
- A função `atoi` converte seu argumento — uma string que inicia com uma série de dígitos que representa um inteiro — em um valor `int`.
- A função `atol` converte seu argumento — uma string que inicia com uma série de dígitos que representa um inteiro long — em um valor `long`.
- A função `strtod` converte uma sequência de caracteres que representa um valor de ponto flutuante em `double`. A função recebe dois argumentos — uma string (`char *`) e o endereço de um ponteiro `char *`. A string contém a sequência de caracteres a ser convertida; e o restante da string depois da conversão é atribuído ao ponteiro para `char *`.
- A função `strtol` converte uma sequência de caracteres que representam um inteiro para `long`. A função recebe três argumentos — uma string (`char *`), o endereço de um ponteiro `char *` e um inteiro. A string contém a sequência de caracteres a ser convertida, ao ponteiro para `char *` é atribuída a localização do primeiro caractere depois do valor convertido e o inteiro especifica a base do valor sendo convertido.
- A função `strtoul` converte uma sequência de caracteres que representa um inteiro em `unsigned long`. A função recebe três argumentos — uma string (`char *`), o endereço de um ponteiro `char *` e um inteiro. A string contém a sequência de caracteres a ser convertida, ao ponteiro para `char *` é atribuída a localização do primeiro caractere depois do valor convertido e o inteiro especifica a base do valor sendo convertido.
- A função `strchr` procura a primeira ocorrência de um caractere em uma string. Se o caractere é localizado, `strchr` retorna um ponteiro para o caractere na string; caso contrário, `strchr` retorna um ponteiro nulo.
- A função `strcspn` determina o comprimento da parte inicial da string em seu primeiro argumento que não contém nenhum caractere da string em seu segundo argumento. A função retorna o comprimento do segmento.
- A função `strpbrk` procura a primeira ocorrência em seu primeiro argumento de qualquer caractere que aparece em seu segundo argumento. Se um caractere do segundo argumento é localizado, `strpbrk` retorna um ponteiro para o caractere; caso contrário, `strpbrk` retorna um ponteiro nulo.
- A função `strrchr` procura a última ocorrência de um caractere em uma string. Se o caractere é localizado, `strrchr` retorna um ponteiro para o caractere na string; caso contrário, retorna um ponteiro nulo.
- A função `strspn` determina o comprimento da parte inicial da string em seu primeiro argumento que só contém caracteres da string em seu segundo argumento e retorna o comprimento do segmento.
- A função `strstr` procura a primeira ocorrência de seu segundo argumento de string em seu primeiro argumento de string. Se a segunda string é localizada na primeira string, um ponteiro para a localização da string no primeiro argumento é retornado; caso contrário, retorna 0.
- A função `memcpy` copia um número especificado de caracteres do objeto para o qual seu segundo argumento aponta no objeto para o qual seu primeiro argumento aponta. A função pode receber um ponteiro para qualquer objeto. Os ponteiros são recebidos por `memcpy` como ponteiros `void` e convertidos em ponteiros `char` para ser utilizados na função. A função `memcpy` manipula os bytes de seu argumento como caracteres.

- A função `memmove` copia um número especificado de bytes do objeto apontado por seu segundo argumento para o objeto apontado por seu primeiro argumento. A cópia é realizada como se os bytes fossem copiados do segundo argumento para um array temporário de caracteres e, então, copiados do array temporário para o primeiro argumento.
- A função `memcmp` compara o número especificado de caracteres de seu primeiro e segundo argumentos.
- A função `memchr` procura a primeira ocorrência de um byte, representado como `unsigned char`, no número especificado de bytes de um objeto. Se o byte é localizado, um ponteiro para ele é retornado; caso contrário, um ponteiro nulo é retornado.
- A função `memset` copia seu segundo argumento, tratado como um `unsigned char`, para um número especificado de bytes do objeto apontado pelo primeiro argumento.

## Terminologia

&, operador E sobre bits	estrutura auto-referencial	nome de estrutura
&=, operador de atribuição E sobre bits	funções de conversão de strings	operadores de atribuição de bits
^, operador de OU exclusivo sobre bits	isalnum	operadores de bits
^=, operador de atribuição OU exclusivo sobre bits	isalpha	strchr
, operador de OU inclusivo sobre bits	isctrl	strcspn
= operador de atribuição OU inclusivo	isdigit	strpbrk
~, operador de complemento de bits	isgraph	strchr
<<, operador de deslocamento para a esquerda	islower	strspn
<<=, operador de atribuição de deslocamento para a esquerda	isprint	strstr
<stdlib>	ispunct	strtod
>>, operador de deslocamento para a direita	isspace	strtol
>>=, operador de atribuição de deslocamento para a direita	isupper	strtoul
atof	isxdigit	struct
atoi	largura de um campo de bit	tipo de dados agregado
atol	máscara	tipo de estrutura
biblioteca de utilitários gerais	memchr	tolower
campo de bit	memcmp	toupper
campo de bit de largura zero	memcpy	typedef
campo de bit não identificado	memmove	
complemento de um preenchimento	memset	

## Exercícios de revisão

**22.1** Preencha as lacunas em cada uma das seguintes sentenças:

- Uma \_\_\_\_\_ é uma coleção de variáveis relacionadas sob um nome.
- Os bits no resultado de uma expressão que utiliza o operador \_\_\_\_\_ são configurados como um se os bits correspondentes em cada operando estiverem configurados como um. Caso contrário, os bits são configurados como zero.
- As variáveis declaradas em uma definição de estrutura são chamadas de seus \_\_\_\_\_.
- Os bits no resultado de uma expressão que utiliza o operador \_\_\_\_\_ são configurados como um se pelo menos um dos bits correspondentes em qualquer operando estiver configurado como um. Caso contrário, os bits são configurados como zero.
- A palavra-chave \_\_\_\_\_ introduz uma declaração de estrutura.
- A palavra-chave \_\_\_\_\_ é utilizada para criar um sinônimo para um tipo de dados previamente definido.
- Cada bit no resultado de uma expressão que utiliza o operador \_\_\_\_\_ é configurado como um se exatamente um dos bits correspondentes em qualquer operando estiver configurado como um. Caso contrário, o bit é configurado como zero.
- O operador E de bits & é frequentemente utilizado para \_\_\_\_\_ bits (isto é, selecionar certos bits de uma string de bits enquanto zera outros).
- Um membro de estrutura é acessado com o operador \_\_\_\_\_ ou \_\_\_\_\_.
- Os operadores \_\_\_\_\_ e \_\_\_\_\_ são utilizados para deslocar os bits de um valor à esquerda ou à direita, respectivamente.

**22.2** Determine se cada uma das seguintes sentenças é verdadeira ou falsa. Se falsa, explique por quê.

- Estruturas podem conter apenas um tipo de dados.
- Membros de estruturas diferentes devem ter nomes únicos.
- A palavra-chave `typedef` é utilizada para definir novos tipos de dados.
- Estruturas são sempre passadas para funções por referência.



- 22.3** Escreva uma única instrução ou um conjunto de instruções para realizar cada uma das seguintes tarefas:
- Defina uma estrutura chamada `Part` contendo a variável `int partNumber` e o array `char partName`, cujos valores podem ser de até 25 caracteres.
  - Defina `PartPtr` como sinônimo do tipo `Part *`.
  - Utilize instruções separadas para declarar a variável `a` como o tipo `Part`, o array `b[ 10 ]` como o tipo `Part` e a variável `ptr` como o ponteiro do tipo para `Part`.
  - Leia um número e um nome do teclado nos membros de variável `a`.
  - Atribua os valores de membro da variável `a` ao elemento três do array `b`.
  - Atribua o endereço do array `b` à variável ponteiro `ptr`.
  - Imprima os valores de membro do elemento três do array `b`, utilizando a variável `ptr` e o operador de ponteiro de estrutura para referenciar os membros.
- 22.4** Localize o erro em cada um dos seguintes exercícios:
- Suponha que `struct Card` tenha sido definido como contendo dois ponteiros para o tipo `char` — a saber, `face` e `suit`. Além disso, a variável `c` foi declarada como o tipo `Card` e a variável `cPtr`, como o ponteiro de tipo para `Card`. O endereço de `c` foi atribuído à variável `cPtr`.
 

```
cout << *cPtr.face << endl;
```
  - Suponha que `struct Card` tenha sido definido como contendo dois ponteiros para o tipo `char` — a saber, `face` e `suit`. Além disso, o array `hearts[ 13 ]` foi declarado como o tipo `Card`. A seguinte instrução deve imprimir o membro `face` do elemento 10 do array.
 

```
cout << hearts.face << endl;
```
  - `struct Person`

```
{
 char lastName[15];
 char firstName[15];
 int age;
}
```
  - Suponha que a variável `p` tenha sido declarada como o tipo `Person`, e a variável `c`, como o tipo `Card`.
 

```
p = c;
```
- 22.5** Escreva uma única instrução para realizar cada uma das seguintes tarefas. Suponha que as variáveis `c` (que armazena um caractere), `x`, `y` e `z` sejam do tipo `int`; as variáveis `d`, `e` e `f` sejam do tipo `double`; a variável `ptr` seja do tipo `char *` e os arrays `s1[ 100 ]` e `s2[ 100 ]` sejam do tipo `char`.
- Converta o caractere armazenado na variável `c` em uma letra maiúscula. Atribua o resultado à variável `c`.
  - Determine se o valor da variável `c` é um dígito. Utilize o operador condicional como mostrado nas figuras 22.18–22.20 para imprimir " `is a` " ou " `is not a` " quando o resultado for exibido.
  - Converta a string "1234567" em `long` e imprima o valor.
  - Determine se o valor da variável `c` é um caractere de controle. Utilize o operador condicional para imprimir " `is a` " ou " `is not a` " quando o resultado for exibido.
  - Atribua a localização da última ocorrência de `c` em `s1` a `ptr`.
  - Converta a string "8.63582" em `double` e imprima o valor.
  - Determine se o valor de `c` é uma letra. Utilize o operador condicional para imprimir " `is a` " ou " `is not a` " quando o resultado for exibido.
  - Atribua a localização da primeira ocorrência de `s2` em `s1` a `ptr`.
  - Determine se o valor de variável `c` é um caractere de impressão. Utilize o operador condicional para imprimir " `is a` " ou " `is not a` " quando o resultado for exibido.
  - Atribua a localização da primeira ocorrência em `s1` de qualquer caractere de `s2` a `ptr`.
  - Atribua a localização da primeira ocorrência de `s2` em `s1` a `ptr`.
  - Converta a string "-21" em `int` e imprima o valor.

## Respostas dos exercícios de revisão

- 22.1** a) estrutura. b) E de bits (&). c) membros. d) OU inclusivo de bits (|). e) `struct`. f) `typedef`. g) OU exclusivo de bits (^). h) mascarar. i) membro de estrutura (.), ponteiro de estrutura (->). j) de deslocamento para a esquerda (<<), de deslocamento para a direita (>>).
- 22.2** a) Falsa. Uma estrutura pode conter muitos tipos de dados.  
b) Falsa. Os membros de estruturas separadas podem ter os mesmos nomes, mas os membros da mesma estrutura devem ter nomes únicos.

- c) Falsa. typedef é utilizado para definir aliases de tipos de dados previamente definidos.  
 d) Falsa. Por padrão, as estruturas são passadas para funções por valor; elas podem ser passadas por referência.

**22.3** a) `struct Part`  
`{`  
`int partNumber;`  
`char partName[ 26 ];`  
`}`;  
 b) `typedef Part * PartPtr;`  
 c) `Part a;`  
`Part b[ 10 ];`  
`Part *ptr;`  
 d) `cin >> a.partNumber >> a.partName;`  
 e) `b[ 3 ] = a;`  
 f) `ptr = b;`  
 g) `cout << ( ptr + 3 )->partNumber << ' '`  
`<< ( ptr + 3 )->partName << endl;`

- 22.4** a) Erro: Os parênteses que devem incluir \*cPtr foram omitidos, fazendo com que a ordem de avaliação da expressão seja incorreta.  
 b) Erro: O subscrito de array foi omitido. A expressão deve ser `hearts[ 10 ].face`.  
 c) Erro: É necessário um ponto-e-vírgula para terminar uma definição de estrutura.  
 d) Erro: As variáveis de diferentes tipos de estrutura não podem ser atribuídas uma à outra.

**22.5** a) `c = toupper( c );`  
 b) `cout << '\\ ' << c << '\\ ' "`  
`<< ( isdigit( c ) ? "is a" : "is not a" )`  
`<< " digit" << endl;`  
 c) `cout << atol( "1234567" ) << endl;`  
 d) `cout << '\\ ' << c << '\\ ' "`  
`<< ( iscntrl( c ) ? "is a" : "is not a" )`  
`<< " control character" << endl;`  
 e) `ptr = strrchr( s1, c );`  
 f) `cout << atof( "8.63582" ) << endl;`  
 g) `cout << '\\ ' << c << '\\ ' "`  
`<< ( isalpha( c ) ? "is a" : "is not a" )`  
`<< " letter" << endl;`  
 h) `ptr = strstr( s1, s2 );`  
 i) `cout << '\\ ' << c << '\\ ' "`  
`<< ( isprint( c ) ? "is a" : "is not a" )`  
`<< " printing character" << endl;`  
 j) `ptr = strpbrk( s1, s2 );`  
 k) `ptr = strchr( s1, c );`  
 l) `cout << atoi( "-21" ) << endl;`

## Exercícios

- 22.6** Forneça a definição de cada uma das seguintes estruturas:
- A estrutura `Inventory`, contendo o array de caracteres `partName[ 30 ]`, o inteiro `partNumber`, o ponto flutuante `price`, o inteiro `stock` e o inteiro `reorder`.
  - Uma estrutura chamada `Address` que contém os arrays de caractere `streetAddress[ 25 ]`, `city[ 20 ]`, `state[ 3 ]` e `zip-Code[ 6 ]`.
  - A estrutura `Student`, que contém os arrays `firstName[ 15 ]` e `lastName[ 15 ]` e a variável `homeAddress` do tipo `struct Address` da parte (b).
  - A estrutura `Test`, que contém 16 campos de bit com larguras de 1 bit. Os nomes dos campos de bit são as letras de a a p.

**22.7** Considere as seguintes definições de estrutura e declarações de variável:

```
struct Customer {
 char lastName[15];
 char firstName[15];
 int customerNumber;
```

```

 struct {
 char phoneNumber[11];
 char address[50];
 char city[15];
 char state[3];
 char zipCode[6];
 } personal;

} customerRecord, *customerPtr;

customerPtr = &customerRecord;

```

Escreva uma expressão separada que acesse os membros de estrutura em cada uma das seguintes partes:

- Membro `lastName` da estrutura `customerRecord`.
- Membro `lastName` da estrutura apontada por `customerPtr`.
- Membro `firstName` da estrutura `customerRecord`.
- Membro `firstName` da estrutura apontada por `customerPtr`.
- Membro `customerNumber` da estrutura `customerRecord`.
- Membro `customerNumber` da estrutura apontada por `customerPtr`.
- Membro `phoneNumber` do membro `personal` da estrutura `customerRecord`.
- Membro `phoneNumber` do membro `personal` da estrutura apontada por `customerPtr`.
- Membro `address` do membro `personal` da estrutura `customerRecord`.
- Membro `address` do membro `personal` da estrutura apontada por `customerPtr`.
- Membro `city` do membro `personal` da estrutura `customerRecord`.
- Membro `city` do membro `personal` da estrutura apontada por `customerPtr`.
- Membro `state` do membro `personal` da estrutura `customerRecord`.
- Membro `state` do membro `personal` da estrutura apontada por `customerPtr`.
- Membro `zipCode` do membro `personal` da estrutura `customerRecord`.
- Membro `zipCode` do membro `personal` da estrutura apontada por `customerPtr`.

**22.8** Modifique o programa da Figura 22.14 para embaralhar as cartas utilizando um embaralhamento de alto desempenho, como mostrado na Figura 22.3. Imprima o baralho resultante no formato de duas colunas, como na Figura 22.4. Preceda cada carta com sua cor.

**22.9** Escreva um programa que desloca 4 bits para a direita uma variável do tipo inteiro. O programa deve imprimir o inteiro em bits antes e depois da operação de deslocamento. Seu sistema coloca zeros ou uns nos bits tornados vagos?

**22.10** Deslocar um inteiro `unsigned` 1 bit para a esquerda é equivalente a multiplicar o valor por 2. Escreva a função `power2` que aceita dois argumentos de inteiro, `number` e `pow` e calcula

$$\text{number} * 2^{\text{pow}}$$

Utilize um operador de deslocamento para calcular o resultado. O programa deve imprimir os valores como inteiros e como bits.

**22.11** O operador de deslocamento para a esquerda pode ser utilizado para empacotar dois valores de caractere em uma variável do tipo inteiro sem sinal de dois bytes. Escreva um programa que insere dois caracteres a partir do teclado e os passa para a função `packCharacters`. Para empacotar dois caracteres em uma variável de inteiro `unsigned`, atribua o primeiro caractere à variável `unsigned`, desloque a variável `unsigned` para a esquerda 8 posições de bits e combine a variável `unsigned` com o segundo caractere utilizando o operador OU inclusivo sobre bits. O programa deve gerar saída dos caracteres no seu formato de bit antes e depois de eles serem empacotados no tipo inteiro `unsigned` para provar que de fato estão empacotados corretamente na variável `unsigned`.

**22.12** Utilizando o operador de deslocamento para a direita, o operador E sobre bits e uma máscara, escreva a função `unpackCharacters` que aceita o inteiro `unsigned` do Exercício 22.11 e o desempacota em dois caracteres. Para desempacotar dois caracteres a partir de um inteiro `unsigned` de dois bytes, combine o inteiro sem sinal com a máscara 65280 (11111111 00000000) e desloque para a direita o resultado de 8 bits. Atribua o valor resultante a uma variável `char`. Então, combine o inteiro `unsigned` com a máscara 255 (00000000 11111111). Atribua o resultado a outra variável `char`. O programa deve imprimir o inteiro `unsigned` em bits antes de ele ser desempacotado e, então, imprimir os caracteres em bits para confirmar que eles foram desempacotados corretamente.

**22.13** Se seu sistema utiliza inteiros de quatro bytes, reescreva o programa do Exercício 22.11 para empacotar quatro caracteres.

**22.14** Se seu sistema utiliza inteiros de quatro bytes, reescreva a função `unpackCharacters` do Exercício 22.12 para desempacotar quatro caracteres. Crie as máscaras necessárias para desempacotar os 4 caracteres deslocando para a esquerda o valor 255 na variável de máscara por 8 bits 0, 1, 2 ou 3 vezes (dependendo do byte que você está desempacotando).

**22.15** Escreva um programa que inverte a ordem dos bits em um valor inteiro `unsigned`. O programa deve inserir o valor do usuário e chamar a função `reverseBits` para imprimir os bits em ordem inversa. Imprima o valor em bits antes e depois de os bits serem invertidos para confirmar que os bits foram corretamente invertidos.

- 22.16** Escreva um programa que demonstra a passagem de um array por valor. [Dica: Utilize uma struct.] Prove que uma cópia foi passada modificando a cópia de array na função chamada.
- 22.17** Escreva um programa que insere um caractere a partir do teclado e testa o caractere com cada função na biblioteca de tratamento de caracteres. Imprima o valor retornado por cada função.
- 22.18** O seguinte programa utiliza a função `multiple` para determinar se o inteiro inserido a partir do teclado é múltiplo de algum inteiro X. Examine a função `multiple` e, então, determine o valor de X.

```

1 // Exercício 22.18: ex22_18.cpp
2 // Este programa determina se um valor é um múltiplo de X.
3 #include <iostream>
4
5 using std::cout;
6 using std::cin;
7 using std::endl;
8
9 bool multiple(int);
10
11 int main()
12 {
13 int y;
14
15 cout << "Enter an integer between 1 and 32000: ";
16 cin >> y;
17
18 if (multiple(y))
19 cout << y << " is a multiple of X" << endl;
20 else
21 cout << y << " is not a multiple of X" << endl;
22
23 return 0;
24 } // fim do main
25
26 // determina se num é um múltiplo de X
27 bool multiple(int num)
28 {
29 bool mult = true;
30
31 for (int i = 0, mask = 1; i < 10; i++, mask <= 1)
32
33 if ((num & mask) != 0) {
34 mult = false;
35 break;
36 } // fim do if
37
38 return mult;
39 } // fim da função multiple

```

- 22.19** O que o seguinte programa faz?

```

1 // Exercício 22.19: ex22_19.cpp
2 #include <iostream>
3
4 using std::cout;
5 using std::cin;

```

```

6 using std::endl;
7 using std::boolalpha;
8
9 bool mystery(unsigned);
10
11 int main()
12 {
13 unsigned x;
14
15 cout << "Enter an integer: ";
16 cin >> x;
17 cout << boolalpha
18 << "The result is " << mystery(x) << endl;
19
20 return 0;
21 } // fim do main
22
23 // O que essa função faz?
24 bool mystery(unsigned bits)
25 {
26 const int SHIFT = 8 * sizeof(unsigned) - 1;
27 const unsigned MASK = 1 << SHIFT;
28 unsigned total = 0;
29
30 for (int i = 0; i < SHIFT + 1; i++, bits <<= 1)
31 {
32 if ((bits & MASK) == MASK)
33 ++total;
34 }
35
36 return !(total % 2);
37 }
38 } // fim da função mystery

```

- 22.20** Escreva um programa que insere uma linha de texto com a função-membro `istream::getline` (como no Capítulo 15) no array de caracteres `s[ 100 ]`. Gere saída da linha em letras maiúsculas e minúsculas.
- 22.21** Escreva um programa que insere quatro strings que representam inteiros, converte as strings em inteiros, soma os valores e imprime o total dos quatro valores. Utilize somente as técnicas de processamento de string no estilo C mostradas neste capítulo.
- 22.22** Escreva um programa que insere quatro strings que representam valores de ponto flutuante, converte as strings em valores `double`, soma os valores e imprime o total dos quatro valores. Utilize somente as técnicas de processamento de string no estilo C mostradas neste capítulo.
- 22.23** Escreva um programa que insere uma linha de texto e uma string de pesquisa a partir do teclado. Utilizando a função `strstr`, localize a primeira ocorrência da string de pesquisa na linha de texto e atribua a localização à variável `searchPtr` do tipo `char *`. Se a string de pesquisa for localizada, imprima o restante da linha de texto começando com a string de pesquisa. Em seguida, utilize `strstr` novamente para localizar a próxima ocorrência da string de pesquisa na linha de texto. Se uma segunda ocorrência for localizada, imprima o restante da linha de texto começando com a segunda ocorrência. [Dica: A segunda chamada para `strstr` deve conter a expressão `searchPtr + 1` como seu primeiro argumento.]
- 22.24** Escreva um programa baseado no programa do Exercício 22.23 que insere várias linhas de texto e uma string de pesquisa e, então, utiliza a função `strstr` para determinar o número total de ocorrências da string nas linhas de texto. Imprima o resultado.
- 22.25** Escreva um programa que insere várias linhas de texto e um caractere de pesquisa e utiliza a função `strchr` para determinar o número total de ocorrências do caractere nas linhas de texto.
- 22.26** Escreva um programa baseado no programa do Exercício 22.25 que insere várias linhas de texto e utiliza a função `strchr` para determinar o número total de ocorrências de cada letra do alfabeto no texto. As letras minúsculas e maiúsculas devem ser contadas juntas. Armazene os totais para cada letra em um array e imprima os valores em formato tabular depois que os totais forem determinados.

- 22.27** O gráfico no Apêndice B mostra as representações numéricas de código para os caracteres no conjunto de caracteres ASCII. Estude esse gráfico e, então, determine se cada uma das seguintes sentenças é *verdadeira* ou *falsa*:
- A letra “A” vem antes da letra “B”.
  - O dígito “9” vem antes do dígito “0”.
  - Os símbolos comumente utilizados para adição, subtração, multiplicação e divisão vêm antes de qualquer dígito.
  - Os dígitos vêm antes das letras.
  - Se um programa de classificação classifica strings na sequência ascendente, então o programa colocará o símbolo de um parêntese direito antes do símbolo de um parêntese esquerdo.

**22.28** Escreva um programa que lê uma série de strings e imprime somente aquelas que iniciam com a letra “b”.

**22.29** Escreva um programa que lê uma série de strings e imprime somente aquelas que terminam com as letras “ED”.

**22.30** Escreva um programa que insere um código ASCII e imprime o caractere correspondente. Modifique esse programa para que ele gere todos os possíveis códigos de três dígitos no intervalo 000–255 e tente imprimir os caracteres correspondentes. O que acontece quando esse programa é executado?

**22.31** Utilizando o gráfico de caracteres ASCII do Apêndice B como um guia, escreva suas próprias versões das funções de tratamento de caracteres da Figura 22.17.

**22.32** Escreva suas próprias versões das funções da Figura 22.21 para converter strings em números.

**22.33** Escreva suas próprias versões das funções da Figura 22.28 para pesquisar strings.

**22.34** Escreva suas próprias versões das funções da Figura 22.35 para manipular blocos de memória.

**22.35** (*Projeto: Um corretor ortográfico*) Muitos pacotes populares de software processador de texto têm verificadores ortográficos integrados. Utilizamos as capacidades de verificação ortográfica ao preparar este livro e descobrimos que, independentemente do cuidado que pensamos dedicar ao escrever um capítulo, a verificação ortográfica do software foi sempre melhor que a nossa revisão manual, que deixava passar alguns erros de ortografia.

Neste projeto, exige-se que você desenvolva seu próprio utilitário de verificação ortográfica. Fazemos sugestões para ajudá-lo a começar. Você então deve considerar a adição de mais capacidades. Você poderia achar útil utilizar um dicionário computadorizado como uma fonte de palavras.

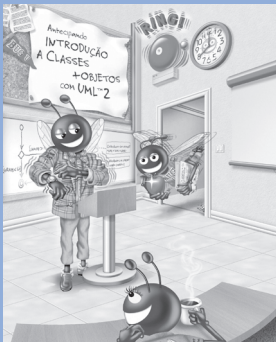
Por que digitamos tantas palavras com ortografia incorreta? Em alguns casos, isso ocorre porque simplesmente não conhecemos a ortografia correta, então fazemos nossa ‘melhor suposição’. Em alguns casos, transpomos duas letras (por exemplo, ‘pardão’ em vez de ‘padrão’). Ocasionalmente digitamos duas vezes uma letra acidentalmente (por exemplo, ‘útil’ em vez de ‘útil’). Às vezes digitamos uma tecla próxima em vez daquela pretendida (por exemplo, ‘amiversário’ em vez de ‘aniversário’). E assim por diante.

Projete e implemente um programa de verificação ortográfica. Seu programa mantém um array `wordList` de strings de caractere. Você pode inserir essas strings ou obtê-las de um dicionário computadorizado.

Seu programa solicita para um usuário inserir uma palavra. O programa então pesquisa essa palavra no array `wordList`. Quando você localizar uma nova palavra que corresponda a uma em `wordList`, imprima essa palavra em uma mensagem como ‘Did you mean “default?”’ [‘Você quis dizer “default?”’].

Se a palavra não estiver presente no array, seu programa deve imprimir ‘Word is not spelled correctly’ [‘A palavra não está grafada corretamente’]. Então seu programa deve tentar localizar outras palavras em `wordList` que talvez sejam a palavra que o usuário pretendeu digitar. Por exemplo, você pode tentar todas as possíveis transposições simples de letras adjacentes para descobrir que a palavra ‘default’ é uma correspondência direta com uma palavra em `wordList`. Naturalmente, isso implica que seu programa verificará todas as outras transposições simples, como ‘edfault’, ‘dfeault’, ‘deafult’, ‘defalut’ e ‘defautl’. Quando você localizar uma nova palavra que corresponda a uma em `wordList`, imprima essa palavra em uma mensagem como ‘Did you mean “default?”’ [‘Você quis dizer “default?”’].

Implemente outros testes, como a substituição de cada letra dupla por uma única letra e alguns outros testes que você pode desenvolver para aprimorar o valor de seu verificador ortográfico.



*As formas que um contêiner  
brilhante pode conter!*  
Theodore Roethke

*Viaje por todo o universo em  
um mapa.*  
Miguel de Cervantes

*Fazes sempre citações  
execráveis; és capaz de  
corromper um santo.*  
William Shakespeare

*Aquela imensa pilha de pó  
chamada 'história'.*  
Augustine Birrell

*O historiador é um profeta ao  
contrário.*  
Friedrich von Schlegel

*Concentre-se em seu objetivo e  
nunca hesite; Nada é tão difícil  
de encontrar se for perseverante  
em sua busca.*  
Robert Herrick

## Standard Template Library (STL)

### OBJETIVOS

Neste capítulo, você aprenderá:

- Como utilizar contêineres de template STL, adaptadores de contêiner e 'semicontêineres'.
- Como programar com as dezenas de algoritmos STL existentes.
- Como os algoritmos utilizam iteradores para acessar os elementos de contêineres STL.
- A familiarizar-se com os recursos STL disponíveis na Internet e World Wide Web.



- 23.1** Introdução à Standard Template Library (STL)
  - 23.1.1** Introdução aos contêineres
  - 23.1.2** Introdução aos iteradores
  - 23.1.3** Introdução aos algoritmos
- 23.2** Contêineres de seqüência
  - 23.2.1** Contêiner de seqüência vector
  - 23.2.2** Contêiner de seqüência list
  - 23.2.3** Contêiner de seqüência deque
- 23.3** Contêineres associativos
  - 23.3.1** Contêiner associativo multiset
  - 23.3.2** Contêiner associativo set
  - 23.3.3** Contêiner associativo multimap
  - 23.3.4** Contêiner associativo map
- 23.4** Adaptadores de contêiner
  - 23.4.1** Adaptador stack
  - 23.4.2** Adaptador queue
  - 23.4.3** Adaptador priority\_queue
- 23.5** Algoritmos
  - 23.5.1** fill, fill\_n, generate e generate\_n
  - 23.5.2** equal, mismatch e lexicographical\_compare
  - 23.5.3** remove, remove\_if, remove\_copy e remove\_copy\_if
  - 23.5.4** replace, replace\_if, replace\_copy e replace\_copy\_if
  - 23.5.5** Algoritmos matemáticos
  - 23.5.6** Algoritmos de pesquisa e classificação básica
  - 23.5.7** swap, iter\_swap e swap\_ranges
  - 23.5.8** copy\_backward, merge, unique e reverse
  - 23.5.9** inplace\_merge, unique\_copy e reverse\_copy
  - 23.5.10** Operações set
  - 23.5.11** lower\_bound, upper\_bound e equal\_range
  - 23.5.12** Heapsort
  - 23.5.13** min e max
  - 23.5.14** Algoritmos de STL não discutidos neste capítulo
- 23.6** Classe bitset
- 23.7** Objetos de função
- 23.8** Síntese
- 23.9** Recursos sobre C++ na Internet e na Web

Resumo | Terminologia | Exercícios de revisão | Respostas dos exercícios de revisão |  
Exercícios | Leitura recomendada

## 23.1 Introdução à Standard Template Library (STL)

Temos enfatizado repetidamente a importância da reutilização de software. Reconhecendo que muitas estruturas de dados e algoritmos são comumente utilizados por programadores em C++, o comitê-padrão do C++ adicionou a [Standard Template Library \(STL\)](#) à C++ Standard Library. A STL define os poderosos componentes reutilizáveis baseados em template que implementam muitas estruturas de dados comuns, e os algoritmos utilizados para processar essas estruturas de dados. A STL oferece prova de conceito para programação genérica com templates — introduzida no Capítulo 14, “Templates”, e demonstrada em detalhes no Capítulo 21, “Estruturas de dados”.

[Nota: Na indústria, os recursos apresentados neste capítulo são comumente referidos como a Standard Template Library ou STL. Entretanto, esses termos não são utilizados no documento-padrão C++, porque esses recursos são simplesmente considerados parte da C++ Standard Library.]

A STL foi desenvolvida por Alexander Stepanov e Meng Lee na Hewlett-Packard e é baseada em sua pesquisa no campo de programação genérica, com contribuições significativas de David Musser. Como você verá, a STL foi concebida e projetada para obter desempenho e flexibilidade.

Este capítulo introduz a STL e discute seus três componentes-chave — os **contêineres** (estruturas de dados baseadas em templates populares), os **iteradores** e os **algoritmos**. Os contêineres STL são estruturas de dados capazes de armazenar objetos de qualquer tipo de dados. Veremos que há três categorias de contêiner — **contêineres de primeira classe**, **adaptadores** e **semicontêineres**.



### Dica de desempenho 23.1

*Para qualquer aplicativo particular, vários contêineres STL diferentes poderiam ser apropriados. Selecione o contêiner mais apropriado que alcance o melhor desempenho (isto é, equilíbrio entre velocidade e tamanho) para esse aplicativo. A eficiência foi uma consideração crucial em design da STL.*



### Dica de desempenho 23.2

*As capacidades da Standard Library são implementadas para operar eficientemente entre muitos aplicativos. Para alguns aplicativos com requisitos de desempenho únicos, talvez seja necessário escrever suas próprias implementações personalizadas.*

Todo contêiner STL tem funções-membro associadas. Um subconjunto dessas funções-membro é definido em todos os contêineres STL. Ilustramos a maior parte dessas funcionalidades comuns em nossos exemplos de contêineres STL **vector** (um array dinamicamente redimensionável que introduzimos no Capítulo 7, “Arrays e vetores”), **list** (uma lista vinculada) e **deque** (uma fila com dupla terminação – *double-ended queue*). Introduzimos as funcionalidades específicas de contêiner nos exemplos de cada um dos outros contêineres STL.

Os iteradores STL, que têm propriedades semelhantes às dos ponteiros, são utilizados por programas para manipular os elementos de contêiner STL. De fato, os arrays-padrão podem ser manipulados como contêineres STL, utilizando ponteiros-padrão como iteradores. Veremos que manipular contêineres com iteradores é conveniente e fornece um poder expressivo radical quando combinado com algoritmos STL — em alguns casos, reduzindo muitas linhas de código a uma única instrução. Há cinco categorias de iteradores, cada uma das quais é discutida na Seção 23.1.2 e utilizada por todo este capítulo.

Os algoritmos STL são funções que realizam manipulações de dados comuns tais como pesquisar, classificar e comparar elementos (ou contêineres inteiros). Aproximadamente 70 algoritmos são implementados no STL. A maioria deles utiliza iteradores para acessar elementos de contêiner. Todo algoritmo tem requisitos mínimos para os tipos de iteradores que podem ser utilizados com ele. Veremos que cada contêiner de primeira classe suporta tipos de iteradores específicos, alguns mais poderosos que outros. O tipo de iterador suportado de um contêiner determina se o contêiner pode ou não ser utilizado com um algoritmo específico. Os iteradores encapsulam o mecanismo utilizado para acessar elementos de contêiner. Esse encapsulamento permite que muitos dos algoritmos de STL sejam aplicados a vários contêineres sem considerar a implementação subjacente do contêiner. Uma vez que os iteradores de um contêiner suportam os requisitos mínimos do algoritmo, então o algoritmo pode processar os elementos desse contêiner. Isso também permite aos programadores criar novos algoritmos que podem processar os elementos de múltiplos tipos de contêiner.



### Observação de engenharia de software 23.1

*A abordagem STL permite que os programas gerais sejam escritos para que o código não dependa do contêiner subjacente. Esse estilo de programação é chamado de programação genérica.*

No Capítulo 21, estudamos as estruturas de dados. Construímos listas vinculadas, filas, pilhas e árvores. Tecemos cuidadosamente objetos vínculo junto com ponteiros. O código baseado em ponteiro é complexo, e a mais leve omissão ou descuido pode levar a sérias violações de acesso de memória e erros de vazamento de memória sem queixas do compilador. Implementar estruturas de dados adicionais, como deque, filas de prioridade, conjuntos e mapas, requer muito trabalho extra. Além disso, se muitos programadores em um grande projeto implementarem contêineres e algoritmos semelhantes para diferentes tarefas, o código torna-se difícil de modificar, manter e depurar. Uma vantagem da STL é que programadores podem reutilizar os contêineres, iteradores e algoritmos STL para implementar representações e manipulações comuns de dados. Essa reutilização pode economizar tempo de desenvolvimento, dinheiro e esforço substanciais.



### Observação de engenharia de software 23.2

*Evite reinventar a roda; programe com os componentes reutilizáveis da C++ Standard Library. A STL inclui muitas das mais populares estruturas de dados como contêineres e fornece vários algoritmos populares para processar dados nesses contêineres.*



## Dica de prevenção de erro 23.1

*Ao programar estruturas de dados baseadas em ponteiro e algoritmos, devemos fazer nossa própria depuração e teste para nos certificarmos de que estruturas de dados, classes e algoritmos funcionam adequadamente. É fácil cometer erros ao manipular ponteiros nesse nível baixo. Vazamentos de memória e violações de acesso de memória são comuns nesse código personalizado. Para a maioria dos programadores e a maioria dos aplicativos que eles precisarão escrever, os contêineres pré-empacotados, 'templatizados' da STL são suficientes. Utilizar a STL ajuda os programadores a reduzir tempo dos testes e da depuração. Uma advertência é que, para projetos grandes, o tempo de compilação de template pode ser significativo.*

Este capítulo faz uma introdução à STL. Ele não é, de modo algum, completo ou abrangente. Entretanto, é um capítulo amigável e acessível que deve convencê-lo do valor da STL na reutilização de software e encorajar mais estudo.

### 23.1.1 Introdução aos contêineres

Os tipos de contêiner STL são mostrados na Figura 23.1. Os contêineres são divididos em três categorias importantes — **contêiner de sequência**, **contêineres associativos** e **adaptadores de contêiner**.

#### Visão geral dos contêineres STL

Os contêineres de sequência (também referidos como **contêineres sequenciais**) representam estruturas de dados lineares, como vetores e listas vinculadas. Os contêineres associativos são contêineres não-lineares que, em geral, podem localizar rapidamente elementos armazenados nos contêineres. Esses contêineres podem armazenar conjuntos de valores ou **pares de chave/valor**. Os contêineres de sequência e contêineres associativos são coletivamente referidos como os contêineres de primeira classe. Como vimos no Capítulo 21, as pilhas e as filas são na realidade versões limitadas de contêineres sequenciais. Por essa razão, a STL implementa pilhas e filas como adaptadores de contêiner que permitem que um programa visualize um contêiner sequencial de maneira limitada. Há outros quatro tipos de contêiner considerados 'semicontêineres' — os arrays baseados em ponteiro no estilo C (discutidos no Capítulo 7), as strings (discutidas no Capítulo 18), os `bitset`s para armazenar conjuntos de valores de flag e `val arrays` para realizar operações matemáticas com vetores em alta velocidade (essa última classe é otimizada para desempenho computacional e não é tão flexível quanto os contêineres de primeira classe). Esses quatro tipos são considerados semicontêineres porque exibem capacidades semelhantes às dos contêineres de primeira classe, mas não suportam todas as capacidades desse tipo de contêiner.

#### Funções comuns do contêiner STL

Todos os contêineres STL fornecem funcionalidades semelhantes. Muitas operações genéricas, como a função-membro `size`, aplicam-se a todos os contêineres, e outras operações se aplicam a subconjuntos de contêineres semelhantes. Isso encoraja a extensibilidade da STL com novas classes. A Figura 23.2 descreve as funções comuns a todos os contêineres Standard Library. [Nota: Os operadores `operator<`, `operator<=`, `operator>`, `operator>=`, `operator==` e `operator!=` sobrecarregados não são oferecidos para `priority_queues`.]

Classe contêiner da Standard Library	Descrição
<i>Contêineres de sequência</i>	
<code>vector</code>	inserções e exclusões rápidas na parte de trás, acesso direto a qualquer elemento
<code>deque</code>	inserções e exclusões rápidas na parte da frente ou de trás, acesso direto a qualquer elemento
<code>list</code>	lista duplamente vinculada, inserção e exclusão rápidas em qualquer parte
<i>Contêineres associativos</i>	
<code>set</code>	pesquisa rápida, não permite duplicatas
<code>multiset</code>	pesquisa rápida, permite duplicatas
<code>map</code>	mapeamento de um para um, permite duplicatas, pesquisa baseada em chave rápida
<code>multimap</code>	mapeamento de um para muitos, permite duplicatas, pesquisa baseada em chave rápida
<i>Adaptadores de contêiner</i>	
<code>stack</code>	último a entrar, primeiro a sair ( <i>last-in, first-out</i> – LIFO)
<code>queue</code>	primeiro a entrar, primeiro a sair ( <i>first-in, first-out</i> – FIFO)
<code>priority_queue</code>	o elemento de maior prioridade é sempre o primeiro elemento

**Figura 23.1** Classes contêineres da Standard Library.

Funções-membro comuns a todos os contêineres STL	Descrição
construtor-padrão	Um construtor para fornecer uma inicialização-padrão do contêiner. Normalmente, cada contêiner tem vários construtores que fornecem diferentes métodos de inicialização para o contêiner.
construtor de cópia	Um construtor que inicializa o contêiner como uma cópia de um contêiner existente do mesmo tipo.
destrutor	Função destrutora para limpeza depois que um contêiner não é mais necessário.
empty	Retorna <code>true</code> se não houver nenhum elemento no contêiner; caso contrário, retorna <code>false</code> .
size	Retorna o número de elementos atualmente no contêiner.
operator=	Atribui um contêiner a outro.
operator<	Retorna <code>true</code> se o primeiro contêiner for menor que o segundo contêiner; caso contrário, retorna <code>false</code> .
operator<=	Retorna <code>true</code> se o primeiro contêiner for menor que ou igual ao segundo contêiner; caso contrário, retorna <code>false</code> .
operator>	Retorna <code>true</code> se o primeiro contêiner for maior que o segundo contêiner; caso contrário, retorna <code>false</code> .
operator>=	Retorna <code>true</code> se o primeiro contêiner for maior que ou igual ao segundo contêiner; caso contrário, retorna <code>false</code> .
operator==	Retorna <code>true</code> se o primeiro contêiner for igual ao segundo contêiner; caso contrário, retorna <code>false</code> .
operator!=	Retorna <code>true</code> se o primeiro contêiner não for igual ao segundo contêiner; caso contrário, retorna <code>false</code> .
swap	Permuta os elementos de dois contêineres.
<i>Funções encontradas apenas em contêineres de primeira classe</i>	
max_size	Retorna o número máximo de elementos para um contêiner.
begin	As duas versões dessa função retornam um <code>iterator</code> ou um <code>const_iterator</code> que referencia o primeiro elemento do contêiner.
end	As duas versões dessa função retornam um <code>iterator</code> ou um <code>const_iterator</code> que referencia a próxima posição depois do fim do contêiner.
rbegin	As duas versões dessa função retornam um <code>reverse_iterator</code> ou um <code>const_reverse_iterator</code> que referencia o último elemento do contêiner.
rend	As duas versões dessa função retornam um <code>reverse_iterator</code> ou um <code>const_reverse_iterator</code> que referencia a próxima posição depois do último elemento do contêiner invertido.
erase	Apaga um ou mais elementos do contêiner.
clear	Apaga todos os elementos do contêiner.

**Figura 23.2** Funções comuns do contêiner STL.

### Arquivos de cabeçalho do contêiner STL

Os arquivos de cabeçalho para cada um dos contêineres Standard Library são mostrados na Figura 23.3. O conteúdo desses arquivos de cabeçalho está inteiramente em namespace `std`.<sup>1</sup>

### typedefs de contêiner de primeira classe comuns

A Figura 23.4 mostra os typedefs comuns (para criar sinônimos ou aliases de nomes de tipo longos) encontrados em contêineres de primeira classe. Esses typedefs são utilizados em declarações genéricas de variáveis, parâmetros para funções e valores de retorno de funções. Por exemplo, o `value_type` em cada contêiner é sempre um typedef que representa o tipo de valor armazenado no contêiner.

<sup>1</sup> Alguns compiladores C++ mais antigos não suportam os arquivos de cabeçalho do novo estilo. Muitos desses compiladores fornecem suas próprias versões dos nomes de arquivo de cabeçalho. Consulte a documentação do compilador para obter informações adicionais sobre o suporte STL fornecido pelo seu compilador.

### Arquivos de cabeçalho de contêiner Standard Library

<code>&lt;vector&gt;</code>	
<code>&lt;list&gt;</code>	
<code>&lt;deque&gt;</code>	
<code>&lt;queue&gt;</code>	Contém tanto <code>queue</code> como <code>priority_queue</code> .
<code>&lt;stack&gt;</code>	
<code>&lt;map&gt;</code>	Contém tanto <code>map</code> como <code>multimap</code> .
<code>&lt;set&gt;</code>	Contém tanto <code>set</code> como <code>multiset</code> .
<code>&lt;bitset&gt;</code>	

**Figura 23.3** Arquivos de cabeçalho de contêiner da Standard Library.

typedef	Descrição
<code>value_type</code>	O tipo de elemento armazenado no contêiner.
<code>reference</code>	Uma referência ao tipo de elemento armazenado no contêiner.
<code>const_reference</code>	Uma referência constante ao tipo de elemento armazenado no contêiner. Essa referência pode ser utilizada somente para <i>leitura</i> de elementos no contêiner e para realizar operações <code>const</code> .
<code>pointer</code>	Um ponteiro para o tipo de elemento armazenado no contêiner.
<code>iterator</code>	Um iterador que aponta para o tipo de elemento armazenado no contêiner.
<code>const_iterator</code>	Um iterador constante que aponta para o tipo de elemento armazenado no contêiner e pode ser utilizado somente para <i>ler</i> elementos.
<code>reverse_iterator</code>	Um iterador invertido que aponta para o tipo de elemento armazenado no contêiner. Esse tipo de iterador serve para iterar por um contêiner na ordem inversa.
<code>const_reverse_iterator</code>	Um iterador invertido constante que aponta para o tipo de elemento armazenado no contêiner e pode ser utilizado somente para <i>ler</i> elementos. Esse tipo de iterador serve para iterar por um contêiner na ordem inversa.
<code>difference_type</code>	O tipo do resultado da subtração de dois iteradores que referenciam o mesmo contêiner (o operador <code>-</code> não é definido para iteradores de <code>lists</code> e contêineres associativos).
<code>size_type</code>	O tipo utilizado para contar itens em um contêiner e em um índice por meio de um contêiner de sequência (não pode indexar por um <code>list</code> ).

**Figura 23.4** typedefs encontrados em contêineres de primeira classe.



### Dica de desempenho 23.3

A *STL*, em geral, evita herança e funções *virtual* em favor do uso da programação genérica com *templates* para alcançar um melhor desempenho em tempo de execução.



### Dica de portabilidade 23.1

A programação com *STL* aprimorará a portabilidade do código.

Ao se preparar para utilizar um contêiner *STL*, é importante assegurar que o tipo de elemento sendo armazenado no contêiner suporta um conjunto mínimo de funcionalidades. Quando um elemento é inserido em um contêiner, uma cópia desse elemento é feita. Por essa razão, o tipo de elemento deve fornecer seu próprio construtor de cópia e seu próprio operador de atribuição. [Nota: Isso só será necessário se a cópia de membro a membro padrão e a atribuição de membro a membro padrão não realizarem as operações de cópia e atribuição adequadas para o tipo de elemento.] Além disso, os contêineres associativos e muitos algoritmos requerem que os elementos sejam comparados. Por essa razão, o tipo de elemento deve fornecer um operador de igualdade (`==`) e um operador menor que (`<`).



## Observação de engenharia de software 23.3

*Tecnicamente, os contêineres STL não requerem que seus elementos sejam comparáveis com os operadores de igualdade e menor que, a menos que um programa utilize uma função-membro de contêiner que deve comparar os elementos do contêiner (por exemplo, a função de classificação na lista de classes). Infelizmente, alguns compiladores C++ pré-padrão não são capazes de ignorar partes de um template que não são utilizadas em um programa particular. Em compiladores com esse problema, talvez você não seja capaz de utilizar os contêineres STL com objetos de classes que não definem os operadores menor que e de igualdade sobrecarregados.*

### 23.1.2 Introdução aos iteradores

Os iteradores têm muitos recursos em comum com ponteiros e são utilizados para apontar para os elementos de contêineres de primeira classe (e para alguns outros propósitos, como veremos). Os iteradores armazenam informações de estado sensíveis aos contêineres particulares em que eles operam; portanto, os iteradores são implementados apropriadamente para cada tipo de contêiner. Certas operações com iteradores são uniformes entre contêineres. Por exemplo, o operador de desreferenciação (\*) desreferencia um iterador a fim de que ele possa utilizar o elemento para o qual aponta. A operação ++ em um iterador move-o para o próximo elemento do contêiner (muito semelhante a como o incremento de um ponteiro em um array visa ao ponteiro para o próximo elemento do array).

Os contêineres STL de primeira classe fornecem as funções-membro `begin` e `end`. A função `begin` retorna um iterador que aponta para o primeiro elemento do contêiner. A função `end` retorna um iterador que aponta para o primeiro elemento após o fim do contêiner (um elemento que não existe). Se o iterador `i` aponta para um elemento particular, então `++i` aponta para o 'próximo' elemento e `*i` referencia o elemento apontado por `i`. O iterador resultante de `end` pode ser utilizado somente em uma comparação de igualdade ou desigualdade para determinar se o 'iterador em movimento' (`i` nesse caso) alcançou o fim do contêiner.

Utilizamos um objeto do tipo `iterator` para referenciar um elemento contêiner que pode ser modificado. Utilizamos um objeto do tipo `const_iterator` para referenciar um elemento contêiner que não pode ser modificado.

Utilizando `istream_iterator` para entrada e o `ostream_iterator` para saída

Utilizamos iteradores com **seqüências** (também chamadas de **intervalos**). Essas seqüências podem estar em contêineres ou em **seqüências de entrada** ou **seqüências de saída**. O programa da Figura 23.5 demonstra a entrada a partir da entrada-padrão (uma seqüência de dados para entrada em um programa), utilizando um `istream_iterator`, e a saída para a saída-padrão (uma seqüência de dados para a saída a partir de um programa), utilizando um `ostream_iterator`. O programa aceita entrada de dois inteiros inseridos pelo usuário a partir do teclado e exibe a soma dos inteiros.<sup>2</sup>

A linha 15 cria um `istream_iterator` que é capaz de extrair (inserir) valores `int` de uma maneira fortemente tipada (*type-safe*) a partir do objeto de entrada-padrão `cin`. A linha 17 desreferencia o iterador `inputInt` para ler o primeiro inteiro de `cin` e atribui esse inteiro a `number1`. Observe que o operador de desreferenciação `*` aplicado a `inputInt` obtém o valor do fluxo associado com `inputInt`; isso é semelhante a desreferenciar um ponteiro. A linha 18 posiciona o iterador `inputInt` para o próximo valor no fluxo de entrada. A linha 19 insere o próximo inteiro de `inputInt` e o atribui a `number2`.

A linha 22 cria um `ostream_iterator` que é capaz de inserir (gerar saída de) valores `int` no objeto de saída-padrão `cout`. A linha 25 gera saída de um inteiro para `cout` atribuindo a `*outputInt` a soma de `number1` e `number2`. Note o uso do operador de desreferenciação `*` para utilizar `*outputInt` como um *lvalue* na instrução de atribuição. Se você quiser gerar saída de outro valor utilizando `outputInt`, o iterador deve ser incrementado com `++` (tanto o incremento prefixado como o incremento pós-fixado podem ser utilizados, mas o estilo prefixo deve ser preferido por razões de desempenho).

```

1 // Figura 23.5: Fig23_05.cpp
2 // Demonstrando entrada e saída com iteradores.
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 #include <iterator> // ostream_iterator e istream_iterator
9
```

Figura 23.5 Iteradores de fluxo de entrada e de saída.

(continua)

<sup>2</sup> Os exemplos neste capítulo precedem cada utilização de uma função STL e cada definição de um objeto contêiner STL com o prefixo `'std::'` em vez de colocar as declarações ou diretivas `using` no início do programa, como foi mostrado na maior parte dos primeiros exemplos. As diferenças em compiladores e o complexo código gerado ao utilizar STL dificultam a construção de um conjunto adequado de declarações ou diretivas `using` que permitam aos programas compilar sem erros. Para permitir que esses programas compilem na ampla variedade de plataformas, escolhemos a abordagem que utiliza o prefixo `'std::'`.



```

10 int main()
11 {
12 cout << "Enter two integers: ";
13
14 // cria istream_iterator para ler valores de int a partir de cin
15 std::istream_iterator< int > inputInt(cin);
16
17 int number1 = *inputInt; // lê int a partir da entrada-padrão
18 ++inputInt; // move iterador para o próximo valor de entrada
19 int number2 = *inputInt; // lê int a partir da entrada-padrão
20
21 // cria ostream_iterator para gravar valores int em cout
22 std::ostream_iterator< int > outputInt(cout);
23
24 cout << "The sum is: ";
25 *outputInt = number1 + number2; // gera saída do resultado para cout
26 cout << endl;
27 return 0;
28 } // fim de main

```

Enter two integers: 12 25  
The sum is: 37

**Figura 23.5** Iteradores de fluxo de entrada e de saída.

(continuação)



### Dica de prevenção de erro 23.2

*O operador \* (de desreferenciação) de qualquer iterador const retorna uma referência const ao elemento contêiner, desativando o uso de funções-membro não-const.*



### Erro comum de programação 23.1

*Tentar desreferenciar um iterador posicionado fora de seu contêiner é um erro de lógica de tempo de execução. Em particular, o iterador retornado por end não pode ser desreferenciado ou incrementado.*



### Erro comum de programação 23.2

*Tentar criar um iterador não-const para um contêiner const resulta em um erro de compilação.*

### Categorias de iterador e a hierarquia de categorias de iterador

A Figura 23.6 mostra as categorias de iteradores utilizados pela STL. Cada categoria fornece um conjunto específico de funcionalidades. A Figura 23.7 ilustra a hierarquia de categorias de iterador. À medida que você segue a hierarquia de cima para baixo, cada categoria de iterador suporta todas as funcionalidades das categorias acima na figura. Assim os tipos de iteradores ‘mais fracos’ estão na parte superior e os mais poderosos estão na parte inferior. Observe que essa não é uma hierarquia de herança.

A categoria de iterador que cada contêiner suporta determina se esse contêiner pode ser utilizado com algoritmos específicos na STL. Os contêineres que suportam iteradores de acesso aleatório podem ser utilizados com todos os algoritmos na STL. Como veremos, os ponteiros para arrays podem ser utilizados no lugar dos iteradores na maioria dos algoritmos STL, incluindo os que requerem iteradores de acesso aleatório. A Figura 23.8 mostra a categoria de iterador de cada um dos contêineres STL. Observe que somente vectors, deque, lists, sets, multisets, maps e multimap (isto é, os contêineres de primeira classe) podem ser percorridos com iteradores.



### Observação de engenharia de software 23.4

*Utilizar o ‘iterador mais fraco’ que produz desempenho aceitável ajuda a produzir componentes maximamente reutilizáveis. Por exemplo, se um algoritmo requer apenas iteradores de leitura direta, ele pode ser utilizado com qualquer contêiner que suporte iteradores de leitura direta, iteradores bidirecionais ou iteradores de acesso aleatório. Entretanto, um algoritmo que requer iteradores de acesso aleatório pode ser utilizado somente com contêineres que tenham iteradores de acesso aleatório.*



Categoria	Descrição
<i>entrada</i>	Utilizado para ler um elemento de um contêiner. Um iterador de entrada pode mover-se somente para a frente (isto é, do início para o final do contêiner) um elemento por vez. Os iteradores de entrada suportam somente algoritmos de uma passagem — o mesmo iterador de entrada não pode ser utilizado para passar por uma sequência duas vezes.
<i>saída</i>	Utilizado para gravar um elemento em um contêiner. Um iterador de saída pode se mover para a frente um elemento por vez. Os iteradores de saída suportam somente algoritmos de uma passagem — o mesmo iterador de saída não pode ser utilizado para passar por uma sequência duas vezes.
<i>leitura direta (forward)</i>	Combina as capacidades dos iteradores de entrada e de saída e retém sua posição no contêiner (como informações de estado).
<i>bidirecional</i>	Combina as capacidades de um iterador de leitura direta com a capacidade de mover-se para trás (isto é, do fim para ao início do contêiner). Os iteradores bidirecionais suportam algoritmos de múltiplas passagens.
<i>acesso aleatório</i>	Combina as capacidades de um iterador bidirecional com a capacidade de acessar diretamente qualquer elemento do contêiner, isto é, pular para a frente ou para trás por um número arbitrário de elementos.

Figura 23.6      Categorias de iterador.

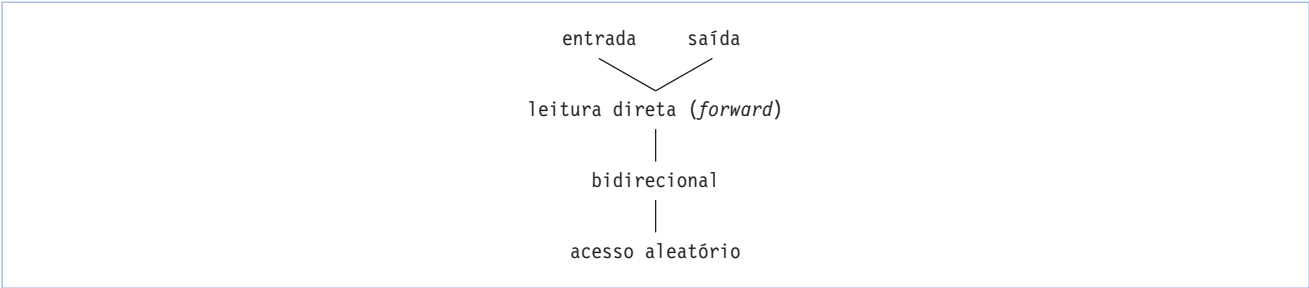


Figura 23.7      Hierarquia de categorias de iterador.

Contêiner	Tipo de iterador suportado
<i>Contêineres de sequência (primeira classe)</i>	
vector	acesso aleatório
deque	acesso aleatório
list	bidirecional
<i>Contêineres associativos (primeira classe)</i>	
set	bidirecional
multiset	bidirecional
map	bidirecional
multimap	bidirecional
<i>Adaptadores de contêiner</i>	
stack	não suporta nenhum iterador
queue	não suporta nenhum iterador
priority_queue	não suporta nenhum iterador

Figura 23.8      Tipos de iterador suportados em cada contêiner Standard Library.

### typedefs de iterador predefinidos

A Figura 23.9 mostra os typedefs de iterador predefinidos que são encontrados nas definições de classe dos contêineres STL. Nem todo typedef é definido para cada contêiner. Utilizamos as versões `const` dos iteradores para percorrer contêineres somente de leitura. Utilizamos iteradores invertidos para percorrer contêineres na direção inversa.



#### Dica de prevenção de erro 23.3

*As operações realizadas em um `const_iterator` retornam referências `const` para impedir modificação em elementos do contêiner sendo manipulado. Utilizar `const_iterators` em preferência a `iterators` onde apropriado é outro exemplo do princípio do menor privilégio.*

### Operações com iteradores

A Figura 23.10 mostra algumas operações que podem ser realizadas em cada tipo de iterador. Observe que as operações para cada tipo de iterador incluem todas as operações que precedem esse tipo na figura. Observe também que, para iteradores de entrada e iteradores de saída, não é possível salvar o iterador e, então, utilizar posteriormente o valor salvo.

#### 23.1.3 Introdução aos algoritmos

A STL fornece algoritmos que podem ser utilizados genericamente em uma variedade de contêineres. A STL fornece muitos algoritmos que você utilizará frequentemente para manipular contêineres. Inserção, exclusão, pesquisa, classificação e outras operações são apropriadas para alguns ou todos os contêineres STL.

A STL inclui aproximadamente 70 algoritmos-padrão. Fornecemos exemplos em ‘código ativo’ (*live-code*) da maioria desses e resumimos outros em tabelas. Os algoritmos operam em elementos contêineres apenas indiretamente por iteradores. Muitos algoritmos operam em seqüências de elementos definidos por pares de iteradores — um primeiro iterador que aponta para o primeiro elemento da seqüência e um segundo iterador que aponta para um elemento um além do último elemento da seqüência. Além disso, é possível criar seus próprios algoritmos novos que operem de modo semelhante para ser utilizados com os contêineres e iteradores STL.

Muitas vezes os algoritmos retornam iteradores que indicam os resultados dos algoritmos. O algoritmo `find`, por exemplo, localiza um elemento e retorna um iterador para esse elemento. Se o elemento não é encontrado, `find` retorna o iterador ‘um além do end’ que foi passado para definir o fim do intervalo a ser pesquisado, o qual pode ser testado para determinar se um elemento não foi encontrado. O algoritmo `find` pode ser utilizado com qualquer contêiner STL de primeira classe. Os algoritmos STL criam ainda outra oportunidade para a reutilização — utilizar a rica coleção de algoritmos populares pode poupar tempo e esforço de programadores.

Se um algoritmo utiliza iteradores menos poderosos, o algoritmo também pode ser utilizado com contêineres que suportam iteradores mais poderosos. Alguns algoritmos demandam iteradores poderosos; por exemplo, `sort` demanda iteradores de acesso aleatório.



#### Observação de engenharia de software 23.5

*A STL é implementada concisamente. Até agora, os designers de classe associavam os algoritmos com os contêineres tornando os algoritmos funções-membro dos contêineres. A STL aceita uma abordagem diferente. Os algoritmos são separados dos contêineres e operam apenas em elementos dos contêineres indiretamente por meio de iteradores. Essa separação facilita a escrita de algoritmos genéricos aplicáveis a muitas classes de contêiner.*



#### Observação de engenharia de software 23.6

*A STL é extensível. É simples e direto adicionar novos algoritmos e isso é feito sem alterações nos contêineres STL.*

typedefs predefinidos para tipos de iterador	Direção de ++	Capacidade
<code>iterator</code>	leitura direta ( <i>forward</i> )	leitura/gravação
<code>const_iterator</code>	leitura direta ( <i>forward</i> )	leitura
<code>reverse_iterator</code>	para trás	leitura/gravação
<code>const_reverse_iterator</code>	para trás	leitura

**Figura 23.9** typedefs de iterador.

Operação com iterador	Descrição
<i>Todos os iteradores</i>	
<code>++p</code>	Pré-incrementa um iterador.
<code>p++</code>	Pós-incrementa um iterador.
<i>Iteradores de entrada</i>	
<code>*p</code>	Desreferencia um iterador.
<code>p = p1</code>	Atribui um iterador a outro.
<code>p == p1</code>	Compara a igualdade de iteradores.
<code>p != p1</code>	Compara a desigualdade de iteradores.
<i>Iteradores de saída</i>	
<code>*p</code>	Desreferencia um iterador.
<code>p = p1</code>	Atribui um iterador para outro.
<i>Iteradores de leitura direta</i>	
Os iteradores de leitura direta fornecem todas as funcionalidades dos iteradores de entrada e dos iteradores de saída.	
<i>Iteradores bidirecionais</i>	
<code>--p</code>	Pré-decrementa um iterador.
<code>p--</code>	Pós-decrementa um iterador.
<i>Iteradores de acesso aleatório</i>	
<code>p += i</code>	Incrementa o iterador <code>p</code> por <code>i</code> posições.
<code>p -= i</code>	Decrementa o iterador <code>p</code> por <code>i</code> posições.
<code>p + i</code>	O valor da expressão é um iterador posicionado em <code>p</code> incrementado por <code>i</code> posições.
<code>p - i</code>	O valor de expressão é um iterador posicionado em <code>p</code> decrementado por <code>i</code> posições.
<code>p[ i ]</code>	Retorna uma referência ao elemento deslocado a partir de <code>p</code> por <code>i</code> posições.
<code>p &lt; p1</code>	Retorna <code>true</code> se o iterador <code>p</code> for menor que o iterador <code>p1</code> (isto é, o iterador <code>p</code> está antes do iterador <code>p1</code> no contêiner); caso contrário, retorna <code>false</code> .
<code>p &lt;= p1</code>	Retorna <code>true</code> se o iterador <code>p</code> for menor que ou igual ao iterador <code>p1</code> (isto é, o iterador <code>p</code> está antes do iterador <code>p1</code> ou na mesma localização que o iterador <code>p1</code> no contêiner); caso contrário, retorna <code>false</code> .
<code>p &gt; p1</code>	Retorna <code>true</code> se o iterador <code>p</code> for maior que o iterador <code>p1</code> (isto é, o iterador <code>p</code> está depois do iterador <code>p1</code> no contêiner); caso contrário, retorna <code>false</code> .
<code>p &gt;= p1</code>	Retorna <code>true</code> se o iterador <code>p</code> for maior que ou igual ao iterador <code>p1</code> (isto é, o iterador <code>p</code> está depois do iterador <code>p1</code> ou na mesma localização que o iterador <code>p1</code> no contêiner); caso contrário, retorna <code>false</code> .

**Figura 23.10** Operações com iteradores para cada tipo de iterador.



## Observação de engenharia de software 23.7

Os algoritmos STL podem operar em contêineres STL e em arrays no estilo C baseados em ponteiros.



## Dica de portabilidade 23.2

Como os algoritmos STL só processam contêineres indiretamente por iteradores, muitas vezes um algoritmo pode ser utilizado com muitos contêineres diferentes.

A Figura 23.11 mostra muitos dos **algoritmos modificadores de seqüência** — isto é, os algoritmos que resultam em modificações nos contêineres aos quais se aplicam.

A Figura 23.12 mostra muitos dos algoritmos não-modificadores de seqüência — isto é, algoritmos que não resultam em modificações nos contêineres aos quais se aplicam. A Figura 23.13 mostra os algoritmos numéricos do arquivo de cabeçalho `<numeric>`.

## 23.2 Contêineres de seqüência

A C++ Standard Template Library fornece três contêineres de seqüência — `vector`, `list` e `deque`. O template da classe `vector` e o template da classe `deque` são ambos baseados em arrays. O template da classe `list` implementa uma estrutura de dados de lista vinculada semelhante à nossa classe `List` apresentada no Capítulo 21, porém mais robusta.

Um dos contêineres mais populares na STL é `vector`. Lembre-se de que introduzimos o template da classe `vector` no Capítulo 7 como um tipo de array mais robusto. Um `vector` muda de tamanho dinamicamente. Diferentemente dos arrays ‘brutos’ do C e C++ (ver o Capítulo 7), `vectors` podem ser atribuídos uns aos outros. Isso não é possível com arrays no estilo C baseados em ponteiros, porque esses nomes de array são ponteiros constantes e não podem ser os alvos de atribuições. Assim como com arrays em C, a subscrição (*subscripting*) de `vector` não realiza verificação automática de intervalos, mas o template da classe `vector` fornece essa capacidade via função-membro `at` (também discutida no Capítulo 7).

Algoritmos modificadores de seqüência		
<code>copy</code>	<code>remove</code>	<code>reverse_copy</code>
<code>copy_backward</code>	<code>remove_copy</code>	<code>rotate</code>
<code>fill</code>	<code>remove_copy_if</code>	<code>rotate_copy</code>
<code>fill_n</code>	<code>remove_if</code>	<code>stable_partition</code>
<code>generate</code>	<code>replace</code>	<code>swap</code>
<code>generate_n</code>	<code>replace_copy</code>	<code>swap_ranges</code>
<code>iter_swap</code>	<code>replace_copy_if</code>	<code>transform</code>
<code>partition</code>	<code>replace_if</code>	<code>unique</code>
<code>random_shuffle</code>	<code>reverse</code>	<code>unique_copy</code>

**Figura 23.11** Algoritmos modificadores de seqüência.

Algoritmos não-modificadores de seqüência		
<code>adjacent_find</code>	<code>find</code>	<code>find_if</code>
<code>count</code>	<code>find_each</code>	<code>mismatch</code>
<code>count_if</code>	<code>find_end</code>	<code>search</code>
<code>equal</code>	<code>find_first_of</code>	<code>search_n</code>

**Figura 23.12** Algoritmos não-modificadores de seqüência.

Algoritmos numéricos provenientes do arquivo de cabeçalho <code>&lt;numeric&gt;</code>	
<code>accumulate</code>	<code>partial_sum</code>
<code>inner_product</code>	<code>adjacent_difference</code>

**Figura 23.13** Algoritmos numéricos provenientes do arquivo de cabeçalho `<numeric>`.

**Dica de desempenho 23.4**

*A inserção na parte posterior de um vector é eficiente. O vector simplesmente cresce, se necessário, para acomodar o novo item. É caro inserir (ou excluir) um elemento no meio de um vector — a parte inteira do vector depois do ponto de inserção (ou exclusão) deve ser movida, porque os elementos de vector ocupam células contíguas na memória da mesma maneira que os arrays ‘brutos’ do C ou C++ ocupam.*

A Figura 23.2 apresentou as operações comuns a todos os contêineres STL. Além dessas operações, todo contêiner normalmente fornece uma variedade de outras capacidades. Muitas dessas capacidades são comuns aos vários contêineres, mas nem sempre são igualmente eficientes para cada contêiner. O programador deve escolher o contêiner mais adequado ao aplicativo.

**Dica de desempenho 23.5**

*Os aplicativos que requerem inserções e exclusões freqüentes em ambas as extremidades de um contêiner normalmente utilizam um deque em vez de um vector. Embora possamos inserir e excluir elementos na parte da frente e na parte de trás de um vector e de um deque, a classe deque é mais eficiente que vector por fazer inserções e exclusões na parte da frente.*

**Dica de desempenho 23.6**

*Os aplicativos com inserções e exclusões freqüentes no meio e/ou nas extremidades de um contêiner normalmente utilizam uma list, devido a sua implementação eficiente de inserção e exclusão em qualquer lugar da estrutura de dados.*

Além das operações comuns descritas na Figura 23.2, os contêineres de seqüência têm várias outras operações comuns — **front** para retornar uma referência ao primeiro elemento do contêiner, **back** para retornar uma referência ao último elemento do contêiner, **push\_back** para inserir um novo elemento no final do contêiner e **pop\_back** para remover o último elemento do contêiner.

**23.2.1 Contêiner de seqüência vector**

O template de classe vector fornece uma estrutura de dados com posições contíguas da memória. Isso permite acesso direto eficiente a qualquer elemento de um vetor via operador de subscrito [], exatamente como com um array ‘bruto’ do C ou C++. O template de classe vector é mais comumente utilizado quando os dados no contêiner precisam ser classificados e facilmente acessíveis via subscrito. Quando a memória de um vector esgotar, o vector aloca uma área contígua maior de memória, copia os elementos originais na nova memória e desaloca a memória antiga.

**Dica de desempenho 23.7**

*Escolha o contêiner vector para obter o melhor desempenho de acesso aleatório.*

**Dica de desempenho 23.8**

*Os objetos do template de classe vector fornecem acesso indexado rápido com o operador de subscrito [] sobrecarregado, porque eles são armazenados na memória contígua como um array bruto C ou C++.*

**Dica de desempenho 23.9**

*É mais rápido inserir muitos elementos de uma vez do que um elemento por vez.*

Uma parte importante de cada contêiner é o tipo de iterador que ele suporta. Isso determina os algoritmos que podem ser aplicados ao contêiner. Um vector suporta iteradores de acesso aleatório — isto é, todas as operações com iteradores mostradas na Figura 23.10 podem ser aplicadas a um iterador vector. Todos os algoritmos STL podem operar em um vector. Os iteradores de um vector normalmente são implementados como ponteiros para elementos do vector. Todo algoritmo STL que aceita argumentos de iterador requer que esses iteradores forneçam um nível mínimo de funcionalidade. Se um algoritmo requer um iterador de leitura direta (*forward*), por exemplo, esse algoritmo pode operar em qualquer contêiner que forneça iteradores de leitura direta, iteradores bidirecionais ou iteradores de acesso aleatório. Contanto que o contêiner suporte a funcionalidade mínima de iterador do algoritmo, este pode operar no contêiner.

**Utilizando vector e iteradores**

A Figura 23.14 ilustra várias funções do template de classe vector. Muitas dessas funções estão disponíveis em todos os contêineres de primeira classe. Você deve incluir o arquivo de cabeçalho <vector> para utilizar o template de classe vector.

A linha 17 define uma instância chamada `integers` do template de classe vector que armazena valores `int`. Quando esse objeto é instanciado, um vector vazio é criado com o tamanho 0 (isto é, o número de elementos armazenado no vector) e capacidade 0 (isto é, o número de elementos que podem ser armazenados sem alocar mais memória para o vector).

As linhas 19 e 20 demonstram as funções `size` e `capacity`; inicialmente, cada uma retorna 0 para o vector `v` nesse exemplo. A função `size` — disponível em cada contêiner — retorna o número de elementos atualmente armazenados no contêiner. A função `capacity` retorna o número de elementos que podem ser armazenados no vector antes de o vector precisar se redimensionar dinamicamente para acomodar mais elementos.

As linhas 23–25 utilizam a função `push_back` — disponível em todos os contêineres de sequência — para adicionar um elemento ao final do vector. Se um elemento é adicionado a um vector cheio, o vector aumenta de tamanho — algumas implementações STL fazem o vector duplicar sua capacidade.

```

1 // Figura 23.14: Fig23_14.cpp
2 // Demonstrando o template de classe vector da Standard Library.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <vector> // definição do template de classe vector
8 using std::vector;
9
10 // protótipo para o template da função printVector
11 template < typename T > void printVector(const vector< T > &integers2);
12
13 int main()
14 {
15 const int SIZE = 6; // define o tamanho do array
16 int array[SIZE] = { 1, 2, 3, 4, 5, 6 }; // inicializa o array
17 vector< int > integers; // cria vector de ints
18
19 cout << "The initial size of integers is: " << integers.size()
20 << "\nThe initial capacity of integers is: " << integers.capacity();
21
22 // função push_back está em cada coleção de sequência
23 integers.push_back(2);
24 integers.push_back(3);
25 integers.push_back(4);
26
27 cout << "\nThe size of integers is: " << integers.size()
28 << "\nThe capacity of integers is: " << integers.capacity();
29 cout << "\n\nOutput array using pointer notation: ";
30
31 // exibe array utilizando a notação de ponteiro
32 for (int *ptr = array; ptr != array + SIZE; ptr++)
33 cout << *ptr << ' ';
34
35 cout << "\n\nOutput vector using iterator notation: ";
36 printVector(integers);
37 cout << "\nReversed contents of vector integers: ";
38
39 // dois iteradores const_reverse
40 vector< int >::const_reverse_iterator reverseIterator;
41 vector< int >::const_reverse_iterator tempIterator = integers.rend();
42
43 // exibe vetor na ordem inversa utilizando reverse_iterator
44 for (reverseIterator = integers.rbegin();
45 reverseIterator != tempIterator; ++reverseIterator)
46 cout << *reverseIterator << ' ';
47

```

Figura 23.14 Template de classe vector da Standard Library.

(continua)

```

48 cout << endl;
49 return 0;
50 } // fim de main
51
52 // template de função para gerar saída de elementos de vector
53 template < typename T > void printVector(const vector< T > &integers2)
54 {
55 typename vector< T >::const_iterator constIterator; // const_iterator
56
57 // exibe elementos vector utilizando const_iterator
58 for (constIterator = integers2.begin();
59 constIterator != integers2.end(); ++constIterator)
60 cout << *constIterator << ' ';
61 } // fim da função printVector

```

```

The initial size of integers is: 0
The initial capacity of integers is: 0
The size of integers is: 3
The capacity of integers is: 4

```

```

Output array using pointer notation: 1 2 3 4 5 6
Output vector using iterator notation: 2 3 4
Reversed contents of vector integers: 4 3 2

```

**Figura 23.14** Template de classe vector da Standard Library.

(continuação)



### Dica de desempenho 23.10

*Pode ser um desperdício duplicar o tamanho de um vector para adicionar mais espaço. Por exemplo, um vector de 1.000.000 elementos se redimensiona para acomodar 2.000.000 elementos quando um novo elemento é adicionado. Isso deixa 999.999 elementos não-utilizados. Os programadores podem utilizar `resize` para controlar o melhor uso do espaço.*

As linhas 27 e 28 utilizam `size` e `capacity` para ilustrar o novo tamanho e a capacidade do vector depois das três operações `push_back`. A função `size` retorna 3 — o número de elementos adicionados ao vector. A função `capacity` retorna 4, indicando que podemos adicionar mais um elemento antes de o vector precisar adicionar mais memória. Quando adicionamos o primeiro elemento, o vector alocou espaço para um elemento, e o tamanho tornou-se 1 para indicar que o vector continha somente um elemento. Quando adicionamos o segundo elemento, a capacidade foi duplicada para 2 e o tamanho também se tornou 2. Quando adicionamos o terceiro elemento, a capacidade é novamente duplicada para 4. Portanto, podemos realmente adicionar outro elemento antes de o vector precisar alocar mais espaço. Quando o vector por fim preencher sua capacidade alocada e o programa tentar adicionar mais um elemento ao vector, o vector dobrará sua capacidade para 8 elementos.

A maneira como um vector cresce para acomodar mais elementos — uma operação demorada — não é especificada pelo C++ Standard Document. Os implementadores de biblioteca C++ utilizam vários esquemas inteligentes para minimizar o overhead de redimensionar um vector. Desse modo, a saída desse programa pode variar, dependendo da versão de vector que acompanha o compilador. Alguns implementadores de biblioteca alocam uma capacidade inicial grande. Se um vector armazena um número pequeno de elementos, essa capacidade pode ser um desperdício de espaço. Entretanto, ela pode melhorar significativamente o desempenho se um programa adicionar muitos elementos a um vector e não tiver de realocar memória para acomodar esses elementos. Essa é uma clássica relação de troca entre espaço e tempo. Os implementadores de biblioteca devem equilibrar a quantidade de memória utilizada com a quantidade de tempo requerida para realizar várias operações com vector.

As linhas 32–33 demonstram como gerar saída do conteúdo de um array utilizando ponteiros e aritmética de ponteiros. A linha 36 chama a função `printVector` (definida nas linhas 53–61) para gerar saída do conteúdo de um vector utilizando iteradores. O template de função `printVector` recebe uma referência `const` a um vector (`integers2`) como seu argumento. A linha 55 define um `const_iterator` chamado `constIterator` que itera pelo vector e gera saída de seu conteúdo. Note que a declaração na linha 55 é prefixada com a palavra-chave `typename`. Como `printVector` é um template de função e `vector< T >` será especializado diferentemente para cada especialização de template de função, o compilador não pode informar em tempo de compilação se `vector< T >::const_iterator` é ou não um tipo. Na especialização particular, `const_iterator` poderia ser uma variável `static`. O compilador precisa dessas informações para compilar o programa corretamente. Portanto, você deve informar ao compilador que um nome qualificado, seja o qualificador ou não um tipo dependente, é esperado como um tipo em cada especialização.



Um `const_iterator` permite ao programa ler os elementos do vector, mas não permite ao programa modificar os elementos. A instrução for nas linhas 58–60 inicializa `constIterator` utilizando a função-membro `vector::begin`, que retorna um `const_iterator` ao primeiro elemento do vector — há outra versão de `begin` que retorna um `iterator` que pode ser utilizado para contêineres não-`const`. Observe que um `const_iterator` é retornado porque o identificador `integers2` foi declarado `const` na lista de parâmetros da função `printVector`. O loop continua contanto que `constIterator` não alcance o fim do vector. Isso é determinado comparando `constIterator` com o resultado de `integers2.end()`, que retorna um `iterator` que indica a localização além do último elemento do vector. Se `constIterator` for igual a esse valor, o fim do vector foi alcançado. As funções `begin` e `end` estão disponíveis para todos os contêineres de primeira classe. O corpo do loop desreferencia o `iterator` `constIterator` para obter o valor no elemento atual do vector. Lembre-se de que o `iterator` atua como um ponteiro para o elemento e que o operador `*` é sobrecarregado para retornar uma referência ao elemento. A expressão `++constIterator` (linha 59) posiciona o `iterator` no próximo elemento do vector.



### Dica de desempenho 23.11

Utilize incremento prefixado quando aplicado a iteradores STL porque o operador de incremento prefixado não retorna um valor que deve ser armazenado em um objeto temporário.



### Dica de prevenção de erro 23.4

Apenas os iteradores de acesso aleatório suportam `<`. É melhor utilizar `!=` e `end` para testar o fim de um contêiner.

A linha 40 declara um `const_reverse_iterator` que pode ser utilizado para iterar por um vector de trás para a frente. A linha 41 declara uma variável `const_reverse_iterator tempIterator` e inicializa o `iterator` retornado pela função `rend` (isto é, o `iterator` para o ponto final ao iterar pelo contêiner na ordem inversa). Todos os contêineres de primeira classe suportam esse tipo de `iterator`. As linhas 44–46 utilizam uma instrução `for` semelhante àquela na função `printVector` para iterar pelo vector. Nesse loop, a função `rbegin` (isto é, o `iterator` para o ponto inicial ao iterar pelo contêiner na ordem inversa) e `tempIterator` delimitam o intervalo de elementos para a saída. Como ocorre com as funções `begin` e `end`, as funções `rbegin` e `rend` podem retornar um `const_reverse_iterator` ou um `reverse_iterator`, com base no fato de o contêiner ser ou não constante.



### Dica de desempenho 23.12

Por razões de desempenho, capture o valor final do loop antes do loop e compare com esse, em vez de fazer uma chamada de função (potencialmente cara) para cada iteração.

## Funções de manipulação de elemento de vector

A Figura 23.15 ilustra as funções que permitem recuperação e manipulação dos elementos de um vector. A linha 17 utiliza um construtor `vector` sobrecarregado que aceita dois `iteradores` como argumentos para inicializar `integers`. Lembre-se de que os ponteiros em um array podem ser utilizados como `iteradores`. A linha 17 inicializa `integers` com o conteúdo de array a partir da localização `array` até — mas não incluindo — a localização `array + SIZE`.

A linha 18 define um `ostream_iterator` chamado `output` que pode ser utilizado para gerar saída de inteiros separados por um caractere de espaço via `cout`. Um `ostream_iterator< int >` é um mecanismo de saída fortemente tipado que gera saída somente de valores do tipo `int` ou de um tipo compatível. O primeiro argumento para o construtor especifica o fluxo de saída e o segundo argumento é uma string que especifica o separador para a saída de valores — nesse caso, a string contém um caractere de espaço em branco. Utilizamos o `ostream_iterator` (definido no cabeçalho `<iterator>`) para gerar saída do conteúdo do vector nesse exemplo.

A linha 21 utiliza o algoritmo `copy` da Standard Library para gerar saída de todo o conteúdo de vector `integers` para a saída-padrão. O algoritmo `copy` copia cada elemento no contêiner que inicia com a localização especificada pelo `iterator` em seu primeiro argumento e continua até — mas não inclui — a localização especificada pelo `iterator` em seu segundo argumento. O primeiro e o segundo argumentos devem satisfazer aos requisitos de `iterator` de entrada — eles devem ser `iteradores` por meio dos quais valores possam ser lidos de um contêiner. Além disso, aplicar `++` ao primeiro `iterator` deve, por fim, fazer com que ele alcance o segundo argumento de `iterator` no contêiner. Os elementos são copiados para a localização especificada pelo `iterator` de saída (isto é, um `iterator` por meio do qual um valor pode ser armazenado ou enviado para a saída) especificado como o último argumento. Nesse caso, o `iterator` de saída é um `ostream_iterator` (`output`) que é anexado a `cout`, desse modo os elementos são copiados para a saída-padrão. Para utilizar os algoritmos da Standard Library, você deve incluir o arquivo de cabeçalho `<algorithm>`.

As linhas 23–24 utilizam as funções `front` e `back` (disponíveis para todos os contêineres de sequência) para determinar o primeiro e o último elementos do vector, respectivamente. Note a diferença entre as funções `front` e `begin`. A função `front` retorna uma referência ao primeiro elemento no vector, enquanto a função `begin` retorna um `iterator` de acesso aleatório que aponta para o primeiro elemento no vector. Note também a diferença entre as funções `back` e `end`. A função `back` retorna uma referência ao último elemento no vector, ao passo que a função `end` retorna um `iterator` de acesso aleatório apontando para o fim do vector (a localização depois do último elemento).

```

1 // Figura 23.15: Fig23_15.cpp
2 // Testando o template de classe vector da Standard Library
3 // Funções de manipulação de elementos.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include <vector> // definição do template de classe vector
9 #include <algorithm> // algoritmo copy
10 #include <iterator> // iterador ostream_iterator
11 #include <stdexcept> // exceção out_of_range
12
13 int main()
14 {
15 const int SIZE = 6;
16 int array[SIZE] = { 1, 2, 3, 4, 5, 6 };
17 std::vector< int > integers(array, array + SIZE);
18 std::ostream_iterator< int > output(cout, " ");
19
20 cout << "Vector integers contains: ";
21 std::copy(integers.begin(), integers.end(), output);
22
23 cout << "\nFirst element of integers: " << integers.front()
24 << "\nLast element of integers: " << integers.back();
25
26 integers[0] = 7; // configura o primeiro elemento como 7
27 integers.at(2) = 10; // configura os elementos nas posições de 2 a 10
28
29 // insere 22 como 2º elemento
30 integers.insert(integers.begin() + 1, 22);
31
32 cout << "\n\nContents of vector integers after changes: ";
33 std::copy(integers.begin(), integers.end(), output);
34
35 // acessa elemento fora do intervalo
36 try
37 {
38 integers.at(100) = 777;
39 } // fim do try
40 catch (std::out_of_range outOfRange) // exceção out_of_range
41 {
42 cout << "\n\nException: " << outOfRange.what();
43 } // fim do catch
44
45 // apaga primeiro elemento
46 integers.erase(integers.begin());
47 cout << "\n\nVector integers after erasing first element: ";
48 std::copy(integers.begin(), integers.end(), output);
49
50 // apaga elementos restantes
51 integers.erase(integers.begin(), integers.end());
52 cout << "\n\nAfter erasing all elements, vector integers "
53 << (integers.empty() ? "is" : "is not") << " empty";
54
55 // insere elementos a partir do array
56 integers.insert(integers.begin(), array, array + SIZE);

```

**Figura 23.15** Funções de manipulação de elemento do template de classe vector.

(continua)

```

57 cout << "\n\nContents of vector integers before clear: ";
58 std::copy(integers.begin(), integers.end(), output);
59
60 // esvazia inteiros; clear chama erase para esvaziar uma coleção
61 integers.clear();
62 cout << "\nAfter clear, vector integers "
63 << (integers.empty() ? "is" : "is not") << " empty" << endl;
64 return 0;
65 } // fim do main

```

Vector integers contains: 1 2 3 4 5 6  
 First element of integers: 1  
 Last element of integers: 6

Contents of vector integers after changes: 7 22 2 10 4 5 6

Exception: invalid vector<T> subscript

Vector integers after erasing first element: 22 2 10 4 5 6  
 After erasing all elements, vector integers is empty

Contents of vector integers before clear: 1 2 3 4 5 6  
 After clear, vector integers is empty

**Figura 23.15** Funções de manipulação de elemento do template de classe vector.

(continuação)



### Erro comum de programação 23.3

*O vector não deve estar vazio; caso contrário, os resultados das funções front e back são indefinidos.*

As linhas 26–27 ilustram duas maneiras de subscrever por meio de um vector (que também pode ser utilizado com os contêineres deque). A linha 26 utiliza o operador de subscrito que é sobrecarregado para retornar uma referência ao valor na localização especificada ou uma referência constante a esse valor, dependendo de o contêiner ser ou não constante. A função `at` (linha 27) realiza a mesma operação, mas com verificação de limites. A função `at` primeiro verifica o valor fornecido como um argumento e determina se ele está nos limites do vector. Se não estiver, a função `at` lança uma exceção `out_of_bounds` definida no cabeçalho `<stdexcept>` (como demonstrado nas linhas 36–43). A Figura 23.16 mostra alguns tipos de exceção STL. (Os tipos de exceção da Standard Library são discutidos no Capítulo 16, “Tratamento de exceções”.)

A linha 30 utiliza uma das três funções **insert** sobrecarregadas fornecidas por cada contêiner de sequência. A linha 30 insere o valor 22 antes do elemento na localização especificada pelo iterador no primeiro argumento. Nesse exemplo, o iterador está apontando para o segundo elemento do vector, portanto 22 é inserido como o segundo elemento e o segundo elemento original torna-se o terceiro elemento do vector. Outras versões de `insert` permitem inserir múltiplas cópias do mesmo valor, iniciando em uma posição particular no contêiner, ou inserir um intervalo de valores de outro contêiner (ou array), iniciando em uma posição particular no contêiner original.

As linhas 46 e 51 utilizam as duas funções **erase** que estão disponíveis em todos os contêineres de primeira classe. A linha 46 indica que o elemento na localização especificada pelo argumento de iterador deve ser removido do contêiner (nesse exemplo, o elemento no início do vector). A linha 51 especifica que todos os elementos no intervalo inicial com a localização do primeiro argumento até — mas não incluindo — a localização do segundo argumento devem ser apagados do contêiner. Nesse exemplo, todos os elementos são apagados do vector. A linha 53 utiliza a função **empty** (disponível para todos os contêineres e adaptadores) para confirmar que o vector está vazio.



### Erro comum de programação 23.4

*Apagar um elemento que contém um ponteiro para um objeto dinamicamente alocado não exclui (delete) esse objeto; isso pode levar a um vazamento de memória.*

A linha 56 demonstra a versão da função `insert` que utiliza o segundo e o terceiro argumentos para especificar a localização inicial e final em uma sequência de valores (possivelmente de outro contêiner; nesse caso, a partir do array de inteiros `array`) que deve ser inserido no vector. Lembre-se de que a localização final especifica a posição na sequência depois do último elemento a ser inserido; a cópia é realizada até — mas não incluindo — essa localização.

Tipos de exceção STL	Descrição
<code>out_of_range</code>	Indica quando o subscrito está fora do intervalo — por exemplo, quando um subscrito inválido é especificado para a função-membro <code>vector::at</code> .
<code>invalid_argument</code>	Indica que um argumento inválido foi passado para uma função.
<code>length_error</code>	Indica uma tentativa de criar um contêiner, <code>string</code> etc. muito longo.
<code>bad_alloc</code>	Indica que uma tentativa de alocar memória com <code>s</code> (ou com um alocador) falhou porque não havia memória suficiente.

**Figura 23.16** Alguns tipos de exceção da STL.

Por fim, a linha 61 utiliza a função `clear` (localizada em todos os contêineres de primeira classe) para esvaziar o `vector`. Essa função chama a versão de `erase` utilizada na linha 51 para esvaziar o `vector`.

[Nota: Outras funções que são comuns a todos os contêineres e comuns a todos os contêineres de sequência ainda não foram abordadas. Discutiremos a maioria dessas funções nas próximas seções. Discutiremos também muitas funções que são específicas a cada contêiner.]

### 23.2.2 Contêiner de sequência `list`

O contêiner de sequência `list` fornece uma implementação eficiente para operações de inserção e exclusão em qualquer localização do contêiner. Se a maioria das inserções e exclusões ocorrer nas extremidades do contêiner, a estrutura de dados `deque` (Seção 23.2.3) fornece uma implementação mais eficiente. O template da classe `list` é implementado como uma lista duplamente vinculada — cada nó na `list` contém um ponteiro para o nó anterior e para o próximo nó na `list`. Isso possibilita que o template de classe `list` suporte iteradores bidirecionais que permitem que o contêiner seja percorrido para a frente e para trás. Todos os algoritmos que requerem iteradores de entrada, de saída, de leitura direta ou bidirecionais podem operar em uma `list`. Muitas das funções-membro `list` manipulam os elementos do contêiner como um conjunto ordenado de elementos.

Além das funções-membro de todos os contêineres STL na Figura 23.2 e das funções-membro comuns de todos os contêineres de sequência discutidas na Seção 23.2, o template da classe `list` fornece outras nove funções-membro — `splice`, `push_front`, `pop_front`, `remove`, `remove_if`, `unique`, `merge`, `reverse` e `sort`. Várias dessas funções-membro são implementações otimizadas por `list` de algoritmos STL apresentados na Seção 23.5. A Figura 23.17 demonstra vários recursos da classe `list`. Lembre-se de que muitas das funções apresentadas nas figuras 23.14–23.15 podem ser utilizadas com a classe `list`. O arquivo de cabeçalho `<list>` deve ser incluído para utilizar classe `list`.

As linhas 18–19 instanciam dois objetos `list` capazes de armazenar inteiros. As linhas 22–23 utilizam a função `push_front` para inserir inteiros no início de `values`. A função `push_front` é específica às classes `list` e `deque` (não a `vector`). As linhas 24–25 utilizam a função `push_back` para inserir inteiros no fim de `values`. Lembre-se de que a função `push_back` é comum a todos os contêineres de sequência.

```

1 // Figura 23.17: Fig23_17.cpp
2 // Programa de teste do template de classe list da biblioteca-padrão.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <list> // definição do template de classe list
8 #include <algorithm> // algoritmo copy
9 #include <iterator> // ostream_iterator
10
11 // protótipo para o template de função printList
12 template < typename T > void printList(const std::list< T > &listRef);
13
14 int main()
15 {
```

**Figura 23.17** Template de classe `list` da Standard Library.

(continua)

```

16 const int SIZE = 4;
17 int array[SIZE] = { 2, 6, 4, 8 };
18 std::list< int > values; // cria lista de ints
19 std::list< int > otherValues; // cria lista de ints
20
21 // insere itens em values
22 values.push_front(1);
23 values.push_front(2);
24 values.push_back(4);
25 values.push_back(3);
26
27 cout << "values contains: ";
28 printList(values);
29
30 values.sort(); // classifica values
31 cout << "\nvalues after sorting contains: ";
32 printList(values);
33
34 // insere elementos do array em otherValues
35 otherValues.insert(otherValues.begin(), array, array + SIZE);
36 cout << "\nAfter insert, otherValues contains: ";
37 printList(otherValues);
38
39 // remove elementos otherValues e insere no fim de values
40 values.splice(values.end(), otherValues);
41 cout << "\nAfter splice, values contains: ";
42 printList(values);
43
44 values.sort(); // classifica values
45 cout << "\nAfter sort, values contains: ";
46 printList(values);
47
48 // insere elementos do array em otherValues
49 otherValues.insert(otherValues.begin(), array, array + SIZE);
50 otherValues.sort();
51 cout << "\nAfter insert, otherValues contains: ";
52 printList(otherValues);
53
54 // remove elementos otherValues e insere em values na ordem classificada
55 values.merge(otherValues);
56 cout << "\nAfter merge:\n values contains: ";
57 printList(values);
58 cout << "\n otherValues contains: ";
59 printList(otherValues);
60
61 values.pop_front(); // remove elemento da parte da frente
62 values.pop_back(); // remove elemento da parte de trás
63 cout << "\nAfter pop_front and pop_back:\n values contains: ";
64 printList(values);
65
66 values.unique(); // remove elementos duplicados
67 cout << "\nAfter unique, values contains: ";
68 printList(values);
69
70 // permuta elementos de values e otherValues
71 values.swap(otherValues);

```

**Figura 23.17** Template de classe `list` da Standard Library.

(continua)

```

72 cout << "\nAfter swap:\n values contains: ";
73 printList(values);
74 cout << "\n otherValues contains: ";
75 printList(otherValues);
76
77 // substitui conteúdo de values por elementos otherValues
78 values.assign(otherValues.begin(), otherValues.end());
79 cout << "\nAfter assign, values contains: ";
80 printList(values);
81
82 // remove elementos otherValues e insere em values na ordem classificada
83 values.merge(otherValues);
84 cout << "\nAfter merge, values contains: ";
85 printList(values);
86
87 values.remove(4); // remove todos os 4s
88 cout << "\nAfter remove(4), values contains: ";
89 printList(values);
90 cout << endl;
91 return 0;
92 } // fim de main
93
94 // definição do template de função printList; utiliza
95 // ostream_iterator e o algoritmo copy para gerar saída de elementos da lista
96 template < typename T > void printList(const std::list< T > &listRef)
97 {
98 if (listRef.empty()) // a lista está vazia
99 cout << "List is empty";
100 else
101 {
102 std::ostream_iterator< T > output(cout, " ");
103 std::copy(listRef.begin(), listRef.end(), output);
104 } // fim de else
105 } // fim da função printList

```

```

values contains: 2 1 4 3
values after sorting contains: 1 2 3 4
After insert, otherValues contains: 2 6 4 8
After splice, values contains: 1 2 3 4 2 6 4 8
After sort, values contains: 1 2 2 3 4 4 6 8
After insert, otherValues contains: 2 4 6 8
After merge:
 values contains: 1 2 2 2 3 4 4 4 6 6 8 8
 otherValues contains: List is empty
After pop_front and pop_back:
 values contains: 2 2 2 3 4 4 4 6 6 8
After unique, values contains: 2 3 4 6 8
After swap:
 values contains: List is empty
 otherValues contains: 2 3 4 6 8
After assign, values contains: 2 3 4 6 8
After merge, values contains: 2 2 3 3 4 4 6 6 8 8
After remove(4), values contains: 2 2 3 3 6 6 8 8

```

**Figura 23.17** Template de classe `list` da Standard Library.

(continuação)

A linha 30 utiliza a função-membro `list::sort` para organizar os elementos na `list` em ordem crescente. [Nota: Isso é diferente da `sort` nos algoritmos STL.] Uma segunda versão da função `sort` que permite ao programador fornecer uma função predicado binária que aceita dois argumentos (valores na lista), realiza uma comparação e retorna um valor `bool` para indicar o resultado. Essa função determina a ordem em que os elementos da `list` são classificados. Essa versão poderia ser particularmente útil para uma `list` que armazena ponteiros em vez de valores. [Nota: Demonstramos uma função predicado unária na Figura 23.28. Uma função predicado unária aceita um único argumento, realiza uma comparação utilizando esse argumento e retorna um valor `bool` indicando o resultado.]

A linha 40 utiliza a função `list::splice` para remover os elementos em `otherValues` e os insere em `values` antes da posição de iterador especificada como o primeiro argumento. Há duas outras versões dessa função. A função `splice` com três argumentos permite que um elemento seja removido do contêiner especificado como o segundo argumento a partir da localização especificada pelo iterador no terceiro argumento. A função `splice` com quatro argumentos utiliza os dois últimos argumentos para especificar um intervalo de localizações que deve ser removido do contêiner no segundo argumento e colocado na localização especificada no primeiro argumento.

Depois de inserir mais elementos em `otherValues` e classificar tanto `values` como `otherValues`, a linha 55 utiliza a função-membro `list::merge` para remover todos os elementos de `otherValues` e inseri-los na ordem classificada em `values`. Ambas as `lists` devem ser classificadas na mesma ordem antes que essa operação seja realizada. Uma segunda versão de `merge` permite ao programador fornecer uma função predicado que aceita dois argumentos (valores na lista) e retorna um valor `bool`. A função predicado especifica a ordem de classificação utilizada por `merge`.

A linha 61 utiliza a função `list::pop_front` para remover o primeiro elemento na `list`. A linha 62 utiliza a função `pop_back` (disponível para todos os contêineres de sequência) para remover o último elemento na `list`.

A linha 66 utiliza a função `list::unique` para remover os elementos duplicados na `list`. A `list` deve estar na ordem classificada (de modo que todas as duplicatas estejam lado a lado) antes de essa operação ser realizada, para garantir que todas as duplicatas sejam eliminadas. Uma segunda versão de `unique` permite ao programador fornecer uma função predicado que aceita dois argumentos (valores na lista) e retorna um valor `bool` para especificar se dois elementos são iguais.

A linha 71 utiliza a função `swap` (disponível para todos os contêineres) para trocar o conteúdo de `values` pelo conteúdo de `otherValues`.

A linha 78 utiliza a função `list::assign` para substituir o conteúdo de `values` pelo conteúdo de `otherValues` no intervalo especificado pelos dois argumentos de iterador. Uma segunda versão de `assign` substitui o conteúdo original por cópias do valor especificado no segundo argumento. O primeiro argumento da função especifica o número de cópias. A linha 87 utiliza a função `list::remove` para excluir todas as cópias do valor 4 da `list`.

### 23.2.3 Contêiner de sequência deque

A classe `deque` fornece muitos dos benefícios de um `vector` e de uma `list` em um contêiner. O termo `deque` é abreviação de ‘double-ended queue’ (fila com dupla terminação). A classe `deque` é implementada para fornecer acesso indexado eficiente (utilizando subscripto) para ler e modificar seus elementos, de modo muito semelhante a um `vector`. A classe `deque` também é implementada para operações eficientes de inserção e exclusão na parte da frente e na parte de trás, de modo muito semelhante ao de uma `list` (embora uma `list` também seja capaz de inserções e exclusões eficientes no meio da `list`). A classe `deque` fornece suporte para iteradores de acesso aleatório, então `deques` podem ser utilizados com todos os algoritmos STL. Uma das utilizações mais comuns de um `deque` é manter uma fila de elementos na ordem primeiro a entrar, primeiro a sair. De fato, um `deque` é a implementação subjacente padrão para o adaptador `queue` (Seção 23.4.2).

O armazenamento adicional de um `deque` pode ser alocado no final do `deque` em blocos de memória que, em geral, são mantidos como um array de ponteiros para esses blocos.<sup>3</sup> Devido ao layout de memória não contígua de um `deque`, um iterador `deque` deve ser mais inteligente que os ponteiros que são utilizados para iterar por `vectors` ou arrays baseados em ponteiro.



#### Dica de desempenho 23.13

*Em geral, deque apresenta um overhead ligeiramente maior que vector.*



#### Dica de desempenho 23.14

*As inserções e exclusões no meio de um deque são otimizadas para minimizar o número de elementos copiados, portanto ele é mais eficiente que um vector, mas menos eficiente que uma list para esse tipo de modificação.*

A classe `deque` fornece as mesmas operações básicas que a classe `vector`, mas adiciona as funções-membro `push_front` e `pop_front` para permitir, respectivamente, inserção e exclusão no início do `deque`.

A Figura 23.18 demonstra os recursos da classe `deque`. Lembre-se de que muitas das funções apresentadas nas figuras 23.14, 23.15 e 23.17 também podem ser utilizadas com a classe `deque`. O arquivo de cabeçalho `<deque>` deve ser incluído para utilizar a classe `deque`.

<sup>3</sup> Esse é um detalhe específico da implementação, não um requisito do padrão C++.



A linha 13 instancia um deque que pode armazenar valores `double`. As linhas 17–19 utilizam as funções `push_front` e `push_back` para inserir elementos no início e no fim do deque. Lembre-se de que `push_back` está disponível para todos os contêineres de sequência, mas `push_front` só está disponível para as classes `list` e `deque`.

A instrução `for` nas linhas 24–25 utiliza o operador de subscrito a fim de recuperar o valor em cada elemento do deque para a saída. Observe que a condição utiliza a função `size` para assegurar que não tentamos acessar um elemento fora dos limites do deque.

A linha 27 utiliza a função `pop_front` para demonstrar a remoção do primeiro elemento do deque. Lembre-se de que `pop_front` está disponível somente para as classes `list` e `deque` (não para a classe `vector`).

A linha 32 utiliza o operador de subscrito para criar um *lvalue*. Isso permite que os valores sejam atribuídos diretamente a qualquer elemento do deque.

```

1 // Figura 23.18: Fig23_18.cpp
2 // Programa de teste da classe deque da Standard Library.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <deque> // definição do template de classe deque
8 #include <algorithm> // algoritmo copy
9 #include <iterator> // ostream_iterator
10
11 int main()
12 {
13 std::deque< double > values; // cria deque de doubles
14 std::ostream_iterator< double > output(cout, " ");
15
16 // insere elementos em values
17 values.push_front(2.2);
18 values.push_front(3.5);
19 values.push_back(1.1);
20
21 cout << "values contains: ";
22
23 // utiliza o operador de subscrito para obter elementos de values
24 for (unsigned int i = 0; i < values.size(); i++)
25 cout << values[i] << ' ';
26
27 values.pop_front(); // remove o primeiro elemento
28 cout << "\nAfter pop_front, values contains: ";
29 std::copy(values.begin(), values.end(), output);
30
31 // utiliza o operador de subscrito para modificar o elemento na localização 1
32 values[1] = 5.4;
33 cout << "\nAfter values[1] = 5.4, values contains: ";
34 std::copy(values.begin(), values.end(), output);
35 cout << endl;
36 return 0;
37 } // fim do main

```

```

values contains: 3.5 2.2 1.1
After pop_front, values contains: 2.2 1.1
After values[1] = 5.4, values contains: 2.2 5.4

```

**Figura 23.18** Template de classe deque da Standard Library.

## 23.3 Contêineres associativos

Os contêineres associativos da STL fornecem acesso direto para armazenar e recuperar elementos via **chaves** (frequentemente chamadas de **chaves de pesquisa**). Os quatro contêineres associativos são `multiset`, `set`, `multimap` e `map`. Cada contêiner associativo mantém suas chaves na ordem classificada. A iteração por um contêiner associativo percorre-o na ordem classificada para esse contêiner. As classes `multiset` e `set` fornecem operações para manipular conjuntos de valores onde os valores são as chaves — não há um valor separado associado a cada chave. A principal diferença entre um `multiset` e um `set` é que um `multiset` permite chaves duplicadas e um `set` não. As classes `multimap` e `map` fornecem operações para manipular valores associados com chaves (esses valores são às vezes referidos como **valores mapeados**). A principal diferença entre um `multimap` e um `map` é que um `multimap` permite que chaves duplicadas com valores associados sejam armazenadas e um `map` permite apenas chaves únicas com valores associados. Além das funções-membro comuns de todos os contêineres apresentados na Figura 23.2, todos os contêineres associativos também suportam várias outras funções-membro, incluindo `find`, `lower_bound`, `upper_bound` e `count`. Os exemplos de cada um dos contêineres associativos e as funções-membro de contêiner associativo comuns são apresentados nas próximas subseções.

### 23.3.1 Contêiner associativo `multiset`

O contêiner associativo `multiset` fornece rápido armazenamento e recuperação de chaves e permite chaves duplicadas. A ordem dos elementos é determinada por um **objeto de função comparador**. Por exemplo, em um `multiset` de inteiros, os elementos podem ser classificados em ordem crescente ordenando as chaves com **objeto de função comparador** `less< int >`. Discutimos objetos de função em detalhes na Seção 23.7. O tipo de dados das chaves em todos os contêineres associativos deve suportar adequadamente a comparação com base no objeto de função comparador especificado — as chaves classificadas com `less< T >` devem suportar a comparação com operador `<`. Se as chaves utilizadas nos contêineres associativos são de tipos de dados definidos pelo usuário, esses tipos devem fornecer os operadores de comparação apropriados. Um `multiset` suporta iteradores bidirecionais (mas não iteradores de acesso aleatório).

A Figura 23.19 demonstra o contêiner associativo `multiset` de um `multiset` de inteiros classificados em ordem crescente. O arquivo de cabeçalho `<set>` deve ser incluído para utilizar a classe `multiset`. Os contêineres `multiset` e `set` fornecem as mesmas funções-membro.

A linha 10 utiliza um `typedef` para criar um novo nome de tipo (alias) para um `multiset` de inteiros organizados em ordem crescente, utilizando o objeto de função `less< int >`. A ordem crescente é o padrão de um `multiset`, então `std::less< int >` pode ser omitido na linha 10. Esse novo tipo (`Im`) é então utilizado para instanciar um objeto `multiset` do tipo inteiro, `intMultiset` (linha 19).

```

1 // Figura 23.19: Fig23_19.cpp
2 // Testando a classe multiset da Standard Library
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <set> // definição do template de classe multiset
8
9 // define o nome abreviado para o tipo multiset utilizado nesse programa
10 typedef std::multiset< int, std::less< int > > Im;
11
12 #include <algorithm> // algoritmo copy
13 #include <iterator> // ostream_iterator
14
15 int main()
16 {
17 const int SIZE = 10;
18 int a[SIZE] = { 7, 22, 9, 1, 18, 30, 100, 22, 85, 13 };
19 Im intMultiset; // Im é o typedef para "multiset integer"
20 std::ostream_iterator< int > output(cout, " ");
21
22 cout << "There are currently " << intMultiset.count(15)
23 << " values of 15 in the multiset\n";
24
25 intMultiset.insert(15); // insere 15 em intMultiset
26 intMultiset.insert(15); // insere 15 em intMultiset

```

Figura 23.19 Template de classe `multiset` da Standard Library.

(continua)

```

27 cout << "After inserts, there are " << intMultiset.count(15)
28 << " values of 15 in the multiset\n\n";
29
30 // iterador que não pode ser utilizado para alterar valores de elemento
31 Ims::const_iterator result;
32
33 // localiza 15 em intMultiset; find retorna iterador
34 result = intMultiset.find(15);
35
36 if (result != intMultiset.end()) // se o iterador não estiver no fim
37 cout << "Found value 15\n"; // localizou valor de pesquisa 15
38
39 // localiza 20 em intMultiset; find retorna iterador
40 result = intMultiset.find(20);
41
42 if (result == intMultiset.end()) // será true daí porque
43 cout << "Did not find value 20\n"; // não localizou 20
44
45 // insere elementos do array a em intMultiset
46 intMultiset.insert(a, a + SIZE);
47 cout << "\nAfter insert, intMultiset contains:\n";
48 std::copy(intMultiset.begin(), intMultiset.end(), output);
49
50 // determina limite inferior e superior de 22 em intMultiset
51 cout << "\n\nLower bound of 22: "
52 << *(intMultiset.lower_bound(22));
53 cout << "\nUpper bound of 22: " << *(intMultiset.upper_bound(22));
54
55 // p representa par de const_iterators
56 std::pair< Ims::const_iterator, Ims::const_iterator > p;
57
58 // usa equal_range para determinar limite inferior e superior
59 // de 22 em intMultiset
60 p = intMultiset.equal_range(22);
61
62 cout << "\n\nequal_range of 22:" << "\n Lower bound: "
63 << *(p.first) << "\n Upper bound: " << *(p.second);
64 cout << endl;
65 return 0;
66 } // fim de main

```

There are currently 0 values of 15 in the multiset  
 After inserts, there are 2 values of 15 in the multiset

Found value 15  
 Did not find value 20

After insert, intMultiset contains:  
 1 7 9 13 15 15 18 22 22 30 85 100

Lower bound of 22: 22  
 Upper bound of 22: 30

equal\_range of 22:  
 Lower bound: 22  
 Upper bound: 30



## Boa prática de programação 23.1

*Utilize typedefs para facilitar a leitura de código com nomes de tipo longos (como multisets).*

A instrução de saída na linha 22 utiliza a função `count` (disponível em todos os contêineres associativos) para contar o número de ocorrências do valor 15 atualmente no `multiset`.

As linhas 25–26 utilizam uma das três versões da função `insert` para adicionar o valor 15 ao `multiset` duas vezes. Uma segunda versão de `insert` aceita um iterador e um valor como argumentos e inicia a procura pelo ponto de inserção a partir da posição especificada. Uma terceira versão de `insert` aceita dois iteradores como argumentos que especificam um intervalo de valores para adicionar ao `multiset` de outro contêiner.

A linha 34 utiliza a função `find` (disponível em todos os contêineres associativos) para localizar o valor 15 no `multiset`. A função `find` retorna um iterador ou um `const_iterator` apontando para a primeira localização em que o valor é encontrado. Se o valor não for encontrado, `find` retorna um iterador ou um `const_iterator` igual ao valor retornado por uma chamada para `end`. A linha 41 demonstra esse caso.

A linha 46 utiliza a função `insert` para inserir os elementos do array `a` no `multiset`. Na linha 48, o algoritmo `copy` copia os elementos do `multiset` para a saída-padrão. Observe que os elementos são exibidos em ordem crescente.

As linhas 52 e 53 utilizam as funções `lower_bound` e `upper_bound` (disponíveis em todos os contêineres associativos) para localizar a primeira ocorrência do valor 22 no `multiset` e o elemento *depois* da última ocorrência do valor 22 no `multiset`. Ambas as funções retornam iteradores ou `const_iterators` que apontam para a localização apropriada ou para o iterador retornado por `end` se o valor não estiver no `multiset`.

A linha 56 instancia uma instância da classe `pair` chamada `p`. Objetos da classe `pair` são utilizados para associar pares de valores. Nesse exemplo, o conteúdo de um `pair` são dois `const_iterators` para nosso `multiset` baseado em inteiro. O propósito de `p` é armazenar o valor de retorno da função `multiset::equal_range`, que retorna um `pair` contendo o resultado de uma operação `lower_bound` e `upper_bound`. O tipo `pair` contém dois membros de dados públicos chamados `first` e `second`.

A linha 60 utiliza a função `equal_range` para determinar `lower_bound` e `upper_bound` de 22 no `multiset`. A linha 63 utiliza `p.first` e `p.second`, respectivamente, para acessar `lower_bound` e `upper_bound`. Desreferenciamos os iteradores para gerar saída dos valores nas localizações retornadas de `equal_range`.

### 23.3.2 Contêiner associativo set

O contêiner associativo `set` é utilizado para rápido armazenamento e recuperação de chaves únicas. A implementação de um `set` é idêntica à de um `multiset`, exceto pelo fato de que um `set` deve ter chaves únicas. Portanto, se uma tentativa de inserir uma chave duplicada for feita em um `set`, a duplicata é ignorada; como esse é o comportamento matemático pretendido de um conjunto, não o identificamos como um erro comum de programação. Um `set` suporta iteradores bidirecionais (mas não iteradores de acesso aleatório). A Figura 23.20 demonstra um `set` de `doubles`. O arquivo de cabeçalho `<set>` deve ser incluído para utilizar a classe `set`.

A linha 10 utiliza `typedef` para criar um novo nome de tipo (`DoubleSet`) para um conjunto de valores `double` organizados em ordem crescente, utilizando o objeto de função `less< double >`.

A linha 19 utiliza o novo tipo `DoubleSet` para instanciar o objeto `doubleSet`. A chamada de construtor aceita os elementos no array `a` entre `a` e `a + SIZE` (isto é, o array inteiro) e os insere no `set`. A linha 23 utiliza o algoritmo `copy` para gerar saída do conteúdo do `set`. Note que o valor 2.1 — que apareceu duas vezes no array `a` — aparece apenas uma vez em `doubleSet`. Isso porque o contêiner `set` não permite duplicatas.

A linha 26 define um `pair` que consiste em um `const_iterator` para um `DoubleSet` e um valor `bool`. Esse objeto armazena o resultado de uma chamada para a função `set::insert`.

A linha 31 utiliza a função `insert` para colocar o valor 13.8 no `set`. O `pair` retornado, `p`, contém um iterador `p.first` apontando para o valor 13.8 no `set` e um valor `bool` que é `true` se o valor foi inserido e `false` se o valor não foi inserido (porque ele já estava no `set`). Nesse caso, 13.8 não estava no conjunto, então foi inserido. A linha 38 tenta inserir 9.5, que já está no `set`. A saída das linhas 39–40 mostra que 9.5 não foi inserido.

### 23.3.3 Contêiner associativo multimap

O contêiner associativo `multimap` é utilizado para rápido armazenamento e recuperação de chaves e valores associados (frequentemente chamados de pares chave/valor). Muitas das funções utilizadas com `multisets` e `sets` são também utilizadas com `multimaps` e `maps`. Os elementos de `multimaps` e `maps` são `pairs` de chaves e valores em vez de valores individuais. Ao inserir em um `multimap` ou `map`, um objeto `pair` que contém a chave e o valor é utilizado. A ordem das chaves é determinada por um objeto de função comparador. Por exemplo, em um `multimap` que utiliza inteiros como o tipo de chave, as chaves podem ser classificadas em ordem crescente ordenando-as com o objeto de função comparador `less< int >`. As chaves duplicadas são permitidas em um `multimap`, portanto múltiplos valores podem ser associados com uma única chave. Isso é frequentemente chamado de relacionamento de um para muitos. Por exemplo, em um sistema de processamento de transações de cartão de crédito, uma conta de cartão de crédito pode ter muitas transações associadas; em uma universidade, um aluno pode fazer muitos cursos, e um professor pode dar aulas para muitos alunos; na hierarquia militar, um posto (como ‘sargento’) tem muitas pessoas. Um `multimap` suporta iteradores bidirecionais, mas não iteradores de acesso aleatório. A Figura 23.21 demonstra o contêiner associativo `multimap`. O arquivo de cabeçalho `<map>` deve ser incluído para utilizar a classe `multimap`.

```

1 // Figura 23.20: Fig23_20.cpp
2 // Programa de teste da classe set da Standard Library.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <set>
8
9 // define nome abreviado para o set type utilizado nesse programa
10 typedef std::set< double, std::less< double > > DoubleSet;
11
12 #include <algorithm>
13 #include <iterator> // ostream_iterator
14
15 int main()
16 {
17 const int SIZE = 5;
18 double a[SIZE] = { 2.1, 4.2, 9.5, 2.1, 3.7 };
19 DoubleSet doubleSet(a, a + SIZE);
20 std::ostream_iterator< double > output(cout, " ");
21
22 cout << "doubleSet contains: ";
23 std::copy(doubleSet.begin(), doubleSet.end(), output);
24
25 // p representa o par contendo const_iterator e bool
26 std::pair< DoubleSet::const_iterator, bool > p;
27
28 // insere 13.8 em doubleSet; insert retorna o par em que
29 // p.first representa a localização de 13.8 em doubleSet e
30 // p.second representa se 13.8 foi inserido
31 p = doubleSet.insert(13.8); // valor não está em set
32 cout << "\n\n" << *(p.first)
33 << (p.second ? " was" : " was not") << " inserted";
34 cout << "\ndoubleSet contains: ";
35 std::copy(doubleSet.begin(), doubleSet.end(), output);
36
37 // insere 9.5 em doubleSet
38 p = doubleSet.insert(9.5); // valor já está em set
39 cout << "\n\n" << *(p.first)
40 << (p.second ? " was" : " was not") << " inserted";
41 cout << "\ndoubleSet contains: ";
42 std::copy(doubleSet.begin(), doubleSet.end(), output);
43 cout << endl;
44 return 0;
45 } // fim de main

```

doubleSet contains: 2.1 3.7 4.2 9.5

13.8 was inserted

doubleSet contains: 2.1 3.7 4.2 9.5 13.8

9.5 was not inserted

doubleSet contains: 2.1 3.7 4.2 9.5 13.8

**Figura 23.20** Template de classe set da Standard Library.

```

1 // Figura 23.21: Fig23_21.cpp
2 // Programa de teste da classe multimap da Standard Library.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <map> // definição do template de classe map
8
9 // define o nome abreviado para o tipo multimap utilizado nesse programa
10 typedef std::multimap< int, double, std::less< int > > Mmid;
11
12 int main()
13 {
14 Mmid pairs; // declara os pares de multimap
15
16 cout << "There are currently " << pairs.count(15)
17 << " pairs with key 15 in the multimap\n";
18
19 // insere dois objetos value_type em pares
20 pairs.insert(Mmid::value_type(15, 2.7));
21 pairs.insert(Mmid::value_type(15, 99.3));
22
23 cout << "After inserts, there are " << pairs.count(15)
24 << " pairs with key 15\n\n";
25
26 // insere cinco objetos value_type em pares
27 pairs.insert(Mmid::value_type(30, 111.11));
28 pairs.insert(Mmid::value_type(10, 22.22));
29 pairs.insert(Mmid::value_type(25, 33.333));
30 pairs.insert(Mmid::value_type(20, 9.345));
31 pairs.insert(Mmid::value_type(5, 77.54));
32
33 cout << "Multimap pairs contains:\nKey\tValue\n";
34
35 // utiliza const_iterator para percorrer elementos de pares
36 for (Mmid::const_iterator iter = pairs.begin();
37 iter != pairs.end(); ++iter)
38 cout << iter->first << '\t' << iter->second << '\n';
39
40 cout << endl;
41 return 0;
42 } // fim de main

```

There are currently 0 pairs with key 15 in the multimap  
 After inserts, there are 2 pairs with key 15

Multimap pairs contains:

Key	Value
5	77.54
10	22.22
15	2.7
15	99.3
20	9.345
25	33.333
30	111.11

**Figura 23.21** Template de classe multimap da Standard Library.



### Dica de desempenho 23.15

*Um `multimap` é implementado para localizar eficientemente todos os valores pareados com uma dada chave.*

A linha 10 utiliza `typedef` para definir o alias `Mmid` para um tipo `multimap` em que o tipo de chave é `int`, o tipo de valor associado de uma chave é `double` e os elementos são colocados em ordem crescente. A linha 14 utiliza o novo tipo para instanciar um `multimap` chamado `pairs`. A linha 16 utiliza a função `count` para determinar o número de pares chave/valor com chave 15.

A linha 20 utiliza a função `insert` para adicionar um novo par chave/valor ao `multimap`. A expressão `Mmid::value_type ( 15, 2.7 )` cria um objeto `pair` em que `first` é a chave (15) de tipo `int` e `second` é o valor (2.7) de tipo `double`. O tipo `Mmid::value_type` é definido como parte do `typedef` para o `multimap`. A linha 21 insere outro objeto `pair` com a chave 15 e o valor 99.3. Então as linhas 23–24 geram saída do número de pares com a chave 15.

As linhas 27–31 inserem cinco `pairs` adicionais no `multimap`. A instrução `for` nas linhas 36–38 gera saída do conteúdo do `multimap`, incluindo tanto as chaves como os valores. A linha 38 utiliza o `const_iterator` chamado `iter` para acessar os membros do `pair` em cada elemento do `multimap`. Note na saída que as chaves aparecem em ordem crescente.

#### 23.3.4 Contêiner associativo map

O contêiner associativo `map` é utilizado para armazenamento rápido e recuperação de chaves únicas e valores associados. As chaves duplicadas não são permitidas em um `map`, desse modo, somente um único valor pode ser associado com cada chave. Isso é chamado **mapeamento de um para um**. Por exemplo, uma empresa que utiliza números de empregado únicos, como 100, 200 e 300, poderia ter um `map` que associa números de empregado com seus ramais de telefone — 4321, 4115 e 5217, respectivamente. Com um `map` você especifica a chave e recupera os dados associados rapidamente. Um `map` é comumente chamado de **array associativo**. Fornecer a chave no operador de subscrito `[]` de um `map` localiza o valor associado com essa chave no `map`. As inserções e exclusões podem ser feitas em qualquer lugar de um `map`.

A Figura 23.22 demonstra o contêiner associativo `map` e utiliza os mesmos recursos da Figura 23.21 para demonstrar o operador de subscrito. O arquivo de cabeçalho `<map>` deve ser incluído para utilizar a classe `map`. As linhas 33 e 34 utilizam o operador de subscrito da classe `map`. Quando o subscrito for uma chave que já está no `map` (linha 33), o operador retorna uma referência ao valor associado. Quando o subscrito for uma chave que não está no `map` (linha 34), o operador insere a chave no `map` e retorna uma referência que pode ser utilizada para associar um valor com essa chave. A linha 33 substitui o valor para a chave 25 (anteriormente 33.333, como especificado na linha 21) por um novo valor, 9999.99. A linha 34 insere um novo `pair` de chave/valor no `map` (o que se chama **criar uma associação**).

```

1 // Figura 23.22: Fig23_22.cpp
2 // Programa de teste da classe map da Standard Library.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <map> // definição do template de classe map
8
9 // define o nome abreviado do tipo map utilizado nesse programa
10 typedef std::map< int, double, std::less< int > > Mid;
11
12 int main()
13 {
14 Mid pairs;
15
16 // insere oito objetos value_type em pairs
17 pairs.insert(Mid::value_type(15, 2.7));
18 pairs.insert(Mid::value_type(30, 111.11));
19 pairs.insert(Mid::value_type(5, 1010.1));
20 pairs.insert(Mid::value_type(10, 22.22));
21 pairs.insert(Mid::value_type(25, 33.333));
22 pairs.insert(Mid::value_type(5, 77.54)); // duplicatas ignoradas
23 pairs.insert(Mid::value_type(20, 9.345));
24 pairs.insert(Mid::value_type(15, 99.3)); // duplicatas ignoradas
25
26 cout << "pairs contains:\nKey\tValue\n";

```

**Figura 23.22** Template de classe `map` da Standard Library.

(continua)



```

27
28 // utiliza const_iterator para percorrer elementos de pares
29 for (Mid::const_iterator iter = pairs.begin();
30 iter != pairs.end(); ++iter)
31 cout << iter->first << '\t' << iter->second << '\n';
32
33 pairs[25] = 9999.99; // utiliza subscripto para alterar o valor da chave 25
34 pairs[40] = 8765.43; // utiliza subscripto para inserir o valor da chave 40
35
36 cout << "\nAfter subscript operations, pairs contains:\nKey\tValue\n";
37
38 // utiliza const_iterator para percorrer elementos de pares
39 for (Mid::const_iterator iter2 = pairs.begin();
40 iter2 != pairs.end(); ++iter2)
41 cout << iter2->first << '\t' << iter2->second << '\n';
42
43 cout << endl;
44 return 0;
45 } // fim do main

```

pairs contains:

Key	Value
5	1010.1
10	22.22
15	2.7
20	9.345
25	33.333
30	111.11

After subscript operations, pairs contains:

Key	Value
5	1010.1
10	22.22
15	2.7
20	9.345
25	9999.99
30	111.11
40	8765.43

Figura 23.22 Template de classe map da Standard Library.

(continuação)

## 23.4 Adaptadores de contêiner

A STL fornece três **adaptadores de contêiner** — `stack`, `queue` e `priority_queue`. Os adaptadores não são contêineres de primeira classe, porque não fornecem a implementação real da estrutura de dados em que os elementos podem ser armazenados e porque os adaptadores não suportam iteradores. O benefício de uma classe adaptadora é que o programador pode escolher uma estrutura de dados subjacente apropriada. As três classes adaptadoras fornecem as funções-membro **push** e **pop** para, respectivamente, inserir e remover adequadamente um elemento em/de cada estrutura de dados adaptadora de maneira apropriada. As próximas subseções fornecem exemplos das classes adaptadoras.

### 23.4.1 Adaptador stack

A classe **stack** permite inserções na e exclusões da estrutura de dados subjacente em uma única extremidade (comumente referida como uma estrutura de dados último a entrar, primeiro a sair). Uma `stack` pode ser implementada com qualquer um dos contêineres de sequência: `vector`, `list` e `deque`. Esse exemplo cria três pilhas de inteiros, utilizando cada um dos contêineres de sequência da Standard Library como a estrutura de dados subjacente para representar a `stack`. Por padrão, uma `stack` é implementada com um `deque`. As operações `stack` são **push** para inserir um elemento na parte superior da `stack` (implementada chamando a função `push_back` do contêiner subjacente), **pop** para remover o elemento superior da `stack` (implementada chamando a função `pop_back` do contêiner subjacente),

**top** para obter uma referência ao elemento superior da stack (implementada chamando a função **back** do contêiner subjacente), **empty** para determinar se a stack está vazia (implementada chamando a função **empty** do contêiner subjacente) e **size** para obter o número de elementos na stack (implementada chamando a função **size** do contêiner subjacente).



### Dica de desempenho 23.16

*Cada uma das operações comuns de uma stack é implementada como uma função inline que chama a função apropriada do contêiner subjacente. Isso evita o overhead de uma segunda chamada de função.*



### Dica de desempenho 23.17

*Para obter o melhor desempenho, utilize a classe **deque** ou **vector** como o contêiner subjacente para um stack.*

A Figura 23.23 demonstra a classe adaptadora stack. O arquivo de cabeçalho **<stack>** deve ser incluído para utilizar classe stack.

As linhas 20, 23 e 26 instanciam três pilhas de inteiro. A linha 20 especifica uma stack de inteiros que utiliza o contêiner **deque** padrão como sua estrutura de dados subjacente. A linha 23 especifica uma stack de inteiros que utiliza um **vector** de inteiros como sua estrutura de dados subjacente. A linha 26 especifica uma stack de inteiros que utiliza um **list** de inteiros como sua estrutura de dados subjacente.

A função **pushElements** (linhas 49–56) insere os elementos em cada stack. A linha 53 utiliza a função **push** (disponível em cada classe adaptadora) para colocar um inteiro na parte superior da stack. A linha 54 utiliza a função **stack top** para recuperar o elemento superior da stack para saída. A função **top** não remove o elemento superior.

A função **popElements** (linhas 59–66) remove os elementos de cada stack. A linha 63 utiliza a função **stack top** para recuperar o elemento superior da stack para saída. A linha 64 utiliza a função **pop** (disponível em cada classe adaptadora) para remover o elemento superior da stack. A função **pop** não retorna um valor.

```

1 // Figura 23.23: Fig23_23.cpp
2 // Programa de teste da classe stack adaptadora da Standard Library.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <stack> // definição de stack adapter
8 #include <vector> // definição do template de classe vector
9 #include <list> // definição do template de classe list
10
11 // protótipo de template de função pushElements
12 template< typename T > void pushElements(T &stackRef);
13
14 // protótipo de template de função popElements
15 template< typename T > void popElements(T &stackRef);
16
17 int main()
18 {
19 // pilha com deque subjacente padrão
20 std::stack< int > intDequeStack;
21
22 // pilha com vetor subjacente
23 std::stack< int, std::vector< int > > intVectorStack;
24
25 // pilha com lista subjacente
26 std::stack< int, std::list< int > > intListStack;
27
28 // insere os valores 0-9 em cada pilha
29 cout << "Pushing onto intDequeStack: ";
30 pushElements(intDequeStack);

```

**Figura 23.23** Classe adaptadora stack da Standard Library.

(continua)

```

31 cout << "\nPushing onto intVectorStack: ";
32 pushElements(intVectorStack);
33 cout << "\nPushing onto intListStack: ";
34 pushElements(intListStack);
35 cout << endl << endl;
36
37 // exibe e remove elementos de cada pilha
38 cout << "Popping from intDequeStack: ";
39 popElements(intDequeStack);
40 cout << "\nPopping from intVectorStack: ";
41 popElements(intVectorStack);
42 cout << "\nPopping from intListStack: ";
43 popElements(intListStack);
44 cout << endl;
45 return 0;
46 } // fim do main
47
48 // insere elementos no objeto stack que stackRef referencia
49 template< typename T > void pushElements(T &stackRef)
50 {
51 for (int i = 0; i < 10; i++)
52 {
53 stackRef.push(i); // insere o elemento na pilha
54 cout << stackRef.top() << ' '; // visualiza (e exibe) elemento superior
55 } // fim do for
56 } // fim da função pushElements
57
58 // remove elementos do objeto stack que stackRef referencia
59 template< typename T > void popElements(T &stackRef)
60 {
61 while (!stackRef.empty())
62 {
63 cout << stackRef.top() << ' '; // visualiza (e exibe) elemento superior
64 stackRef.pop(); // remove elemento superior
65 } // fim do while
66 } // fim da função popElements

```

```

Pushing onto intDequeStack: 0 1 2 3 4 5 6 7 8 9
Pushing onto intVectorStack: 0 1 2 3 4 5 6 7 8 9
Pushing onto intListStack: 0 1 2 3 4 5 6 7 8 9

```

```

Popping from intDequeStack: 9 8 7 6 5 4 3 2 1 0
Popping from intVectorStack: 9 8 7 6 5 4 3 2 1 0
Popping from intListStack: 9 8 7 6 5 4 3 2 1 0

```

Figura 23.23 Classe adaptadora stack da Standard Library.

(continuação)

### 23.4.2 Adaptador queue

A classe **queue** permite inserções na parte posterior da estrutura de dados subjacente e exclusões a partir da frente (comumente referido como uma estrutura de dados primeiro a entrar, primeiro a sair). Uma queue pode ser implementada com a estrutura de dados STL **list** ou deque. Por padrão, uma queue é implementada com um deque. As operações queue comuns são **push** para inserir um elemento na parte de trás de queue (implementada chamando a função **push\_back** do contêiner subjacente), **pop** para remover o elemento na frente de queue (implementada chamando a função **pop\_front** do contêiner subjacente), **front** para obter uma referência ao primeiro elemento de queue (implementada chamando a função **front** do contêiner subjacente), **back** para obter uma referência ao último elemento de queue (implementada chamando a função **back** do contêiner subjacente), **empty** para determinar se a queue está vazia (implementada chamando a função **empty** do contêiner subjacente) e **size** para obter o número de elementos em queue (implementada chamando a função **size** do contêiner subjacente).

**Dica de desempenho 23.18**

*Cada uma das operações comuns de uma queue é implementada como uma função inline que chama a função apropriada do contêiner subjacente. Isso evita o overhead de uma segunda chamada de função.*

**Dica de desempenho 23.19**

*Para obter o melhor desempenho, utilize a classe deque ou list como o contêiner subjacente para um queue.*

A Figura 23.24 demonstra a classe adaptadora queue. O arquivo de cabeçalho `<queue>` deve ser incluído para utilizar uma queue.

A linha 11 instancia uma queue que armazena valores `double`. As linhas 14–16 utilizam a função `push` para adicionar elementos a queue. A instrução `while` nas linhas 21–25 utiliza a função `empty` (disponível em todos os contêineres) para determinar se a queue está vazia (linha 21). Enquanto houver mais elementos na queue, a linha 23 utiliza a função `queue front` para ler (mas não remover) o primeiro elemento na queue para saída. A linha 24 remove o primeiro elemento na queue com a função `pop` (disponível em todas as classes adaptadoras).

**23.4.3 Adaptador priority\_queue**

A classe `priority_queue` fornece funcionalidades que permitem inserções na ordem de classificação na estrutura de dados subjacente e exclusões a partir da frente da estrutura de dados subjacente. Uma `priority_queue` pode ser implementada com os contêineres de sequência STL `vector` ou `deque`. Por padrão, uma `priority_queue` é implementada com um `vector` como o contêiner subjacente. Ao adicionar elementos a uma `priority_queue`, eles são inseridos na ordem de prioridade de tal modo que o elemento de maior prioridade (isto é, o maior valor) será o primeiro elemento removido da `priority_queue`. Isso normalmente é realizado via uma técnica de classificação chamada **heapsort** que sempre mantém o maior valor (isto é, elemento de maior prioridade) na frente da estrutura de dados

```

1 // Figura 23.24: Fig23_24.cpp
2 // Programa de teste da fila adaptadora da Standard Library.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <queue> // definição da classe queue adaptadora
8
9 int main()
10 {
11 std::queue< double > values; // fila com doubles
12
13 // insere elementos nos valores de fila
14 values.push(3.2);
15 values.push(9.8);
16 values.push(5.4);
17
18 cout << "Popping from values: ";
19
20 // remove elementos da fila
21 while (!values.empty())
22 {
23 cout << values.front() << ' '; // visualiza elemento da frente da fila
24 values.pop(); // remove o elemento
25 } // fim do while
26
27 cout << endl;
28 return 0;
29 } // fim do main

```

Popping from values: 3.2 9.8 5.4

**Figura 23.24** Templates de classe adaptadora queue da Standard Library.

— tal estrutura de dados é chamada de **heap**. A comparação de elementos é realizada com objeto de função comparador `less< T >` por padrão, mas o programador pode fornecer um comparador diferente.

As operações `priority_queue` comuns são **push** para inserir um elemento na localização apropriada com base na ordem de prioridade de `priority_queue` (implementada chamando a função `push_back` do contêiner subjacente e, então, reordenando os elementos com `heapsort`), **pop** para remover o elemento de maior prioridade de `priority_queue` (implementada chamando a função `pop_back` do contêiner subjacente depois de remover o elemento superior do `heap`), **top** para obter uma referência ao elemento superior do `priority_queue` (implementada chamando a função `front` do contêiner subjacente), **empty** para determinar se o `priority_queue` está ou não vazio (implementada chamando a função `empty` do contêiner subjacente) e **size** para obter o número de elementos na `priority_queue` (implementada chamando a função `size` do contêiner subjacente).



### Dica de desempenho 23.20

*Cada uma das operações comuns de uma `priority_queue` é implementada como uma função inline que chama a função apropriada do contêiner subjacente. Isso evita o overhead de uma segunda chamada de função.*



### Dica de desempenho 23.21

*Para obter o melhor desempenho, utilize a classe `vector` ou `deque` como o contêiner subjacente para um `priority_queue`.*

A Figura 23.25 demonstra a classe adaptadora `priority_queue`. O arquivo de cabeçalho `<queue>` deve ser incluído para utilizar a classe `priority_queue`. A linha 11 instancia uma `priority_queue` que armazena os valores `double` e utiliza um `vector` como a estrutura de dados subjacente. As linhas 14–16 utilizam a função `push` para adicionar elementos a `priority_queue`. A instrução `while` nas linhas 21–25 utiliza a função `empty` (disponível em todos os contêineres) para determinar se a `priority_queue` está vazia (linha 21).

```

1 // Figura 23.25: Fig23_25.cpp
2 // Programa de teste priority_queue adaptador da Standard Library.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <queue> // definição do adaptador priority_queue
8
9 int main()
10 {
11 std::priority_queue< double > priorities; // cria priority_queue
12
13 // insere elementos em prioridades
14 priorities.push(3.2);
15 priorities.push(9.8);
16 priorities.push(5.4);
17
18 cout << "Popping from priorities: ";
19
20 // remove elemento de priority_queue
21 while (!priorities.empty())
22 {
23 cout << priorities.top() << ' '; // visualiza o elemento superior
24 priorities.pop(); // remove o elemento superior
25 } // fim do while
26
27 cout << endl;
28 return 0;
29 } // fim do main

```

Popping from priorities: 9.8 5.4 3.2

**Figura 23.25** Classe adaptadora `priority_queue` da Standard Library.

Enquanto houver mais elementos, a linha 23 utiliza a função `priority_queue::top` para recuperar o elemento de maior prioridade na `priority_queue` para saída. A linha 24 remove o elemento de maior prioridade na `priority_queue` com a função `pop` (disponível em todas as classes adaptadoras).

## 23.5 Algoritmos

Antes da STL, as bibliotecas de classes de contêineres e algoritmos eram essencialmente incompatíveis entre fornecedores. As primeiras bibliotecas de contêiner utilizavam herança e polimorfismo, com o overhead associado de chamadas de função virtual. As primeiras bibliotecas construíam os algoritmos nas classes de contêiner como comportamentos de classe. A STL separa os algoritmos dos contêineres. Isso torna a adição de novos algoritmos muito mais fácil. Com a STL, os elementos de contêineres são acessados por iteradores. As próximas subseções demonstram muitos dos algoritmos STL.



### Dica de desempenho 23.22

*A STL é implementada para eficiência. Ela evita o overhead de chamadas de função virtual.*



### Observação de engenharia de software 23.8

*Os algoritmos STL não dependem dos detalhes da implementação dos contêineres em que eles operam. Contanto que os iteradores do contêiner (ou do array) satisfaçam aos requisitos do algoritmo, os algoritmos STL podem funcionar em arrays baseados em ponteiros no estilo C, em contêineres STL e em estrutura de dados definida pelo usuário.*



### Observação de engenharia de software 23.9

*Os algoritmos podem ser facilmente adicionados à STL sem modificar as classes de contêiner.*

#### 23.5.1 `fill`, `fill_n`, `generate` e `generate_n`

A Figura 23.26 demonstra os algoritmos `fill`, `fill_n`, `generate` e `generate_n`. As funções `fill` e `fill_n` configuram cada elemento em um intervalo de elementos de contêiner com um valor específico. As funções `generate` e `generate_n` utilizam uma **função geradora** para criar valores para cada elemento em um intervalo de elementos de contêiner. A função geradora não aceita nenhum argumento e retorna um valor que pode ser colocado em um elemento do contêiner.

A linha 15 define um vetor de 10 elementos que armazena valores `char`. A linha 17 utiliza a função `fill` para colocar o caractere '5' em cada elemento do vetor `chars` de `chars.begin()` até, mas não incluindo, `chars.end()`. Observe que os iteradores fornecidos como o primeiro e o segundo argumentos devem ser pelo menos iteradores de leitura direta (*forward*) (isto é, podem ser utilizados tanto para entrada a partir de um contêiner como para saída para um contêiner na direção para a frente).

A linha 23 utiliza a função `fill_n` para colocar o caractere 'A' nos primeiros cinco elementos de vetor `chars`. O iterador fornecido como o primeiro argumento deve ser pelo menos um iterador de saída (isto é, pode ser utilizado para saída para um contêiner na direção para a frente). O segundo argumento especifica o número de elementos a ser preenchido. O terceiro argumento especifica o valor a ser colocado em cada elemento.

A linha 29 utiliza a função `generate` para colocar o resultado de uma chamada para função geradora `nextLetter` em cada elemento de vetor `chars` de `chars.begin()` até, mas não incluindo, `chars.end()`. Os iteradores fornecidos como o primeiro e o segundo argumentos devem ser pelo menos iteradores de leitura direta. A função `nextLetter` (definida nas linhas 45–49) inicia com o caractere 'A' mantido em uma variável local `static`. A instrução na linha 48 pós-incrementa o valor de `letter` e retorna o valor antigo de `letter` toda vez que `nextLetter` é chamado.

A linha 35 utiliza a função `generate_n` para colocar o resultado de uma chamada à função geradora `nextLetter` em cinco elementos vetor `chars`, iniciando de `chars.begin()`. O iterador fornecido como o primeiro argumento deve ser pelo menos um iterador de saída.

```
1 // Figura 23.26: Fig23_26.cpp
2 // Algoritmos fill, fill_n, generate e generate_n da Standard Library.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
```

**Figura 23.26** Os algoritmos `fill`, `fill_n`, `generate` e `generate_n`.

(continua)

```

7 #include <algorithm> // definições de algoritmo
8 #include <vector> // definição do template de classe vector
9 #include <iterator> // ostream_iterator
10
11 char nextLetter(); // protótipo de função geradora
12
13 int main()
14 {
15 std::vector< char > chars(10);
16 std::ostream_iterator< char > output(cout, " ");
17 std::fill(chars.begin(), chars.end(), '5'); // preenche chars com 5s
18
19 cout << "Vector chars after filling with 5s:\n";
20 std::copy(chars.begin(), chars.end(), output);
21
22 // preenche os primeiros cinco elementos de chars com As
23 std::fill_n(chars.begin(), 5, 'A');
24
25 cout << "\n\nVector chars after filling five elements with As:\n";
26 std::copy(chars.begin(), chars.end(), output);
27
28 // gera valores para todos os elementos de chars com nextLetter
29 std::generate(chars.begin(), chars.end(), nextLetter);
30
31 cout << "\n\nVector chars after generating letters A-J:\n";
32 std::copy(chars.begin(), chars.end(), output);
33
34 // gera valores para os primeiros cinco elementos de chars com nextLetter
35 std::generate_n(chars.begin(), 5, nextLetter);
36
37 cout << "\n\nVector chars after generating K-O for the"
38 << " first five elements:\n";
39 std::copy(chars.begin(), chars.end(), output);
40 cout << endl;
41 return 0;
42 } // fim de main
43
44 // função geradora retorna próxima letra (inicia com A)
45 char nextLetter()
46 {
47 static char letter = 'A';
48 return letter++;
49 } // fim da função nextLetter

```

Vector chars after filling with 5s:  
5 5 5 5 5 5 5 5 5

Vector chars after filling five elements with As:  
A A A A A 5 5 5 5 5

Vector chars after generating letters A-J:  
A B C D E F G H I J

Vector chars after generating K-O for the first five elements:  
K L M N O F G H I J

**Figura 23.26** Os algoritmos fill, fill\_n, generate e generate\_n.

(continuação)



### 23.5.2 equal, mismatch e lexicographical\_compare

A Figura 23.27 demonstra seqüências de comparação de valores quanto a igualdade utilizando os algoritmos `equal`, `mismatch` e `lexicographical_compare`.

A linha 29 utiliza a função `equal` para comparar a igualdade de duas seqüências de valores. Cada seqüência não precisa necessariamente conter o mesmo número de elementos — `equal` retorna `false` se as seqüências não tiverem o mesmo comprimento. O operador `==` (predefinido ou sobrecarregado) realiza a comparação dos elementos. Nesse exemplo, os elementos em `vector v1` a partir de `v1.begin()` até, mas não incluindo, `v1.end()` são comparados como elementos em `vector v2` iniciando de `v2.begin()`. Nesse exemplo, `v1` e `v2` são iguais. Os três argumentos de iterador devem ser pelo menos iteradores de entrada (isto é, podem ser utilizados para entrada a partir de uma seqüência direta para frente). A linha 34 utiliza a função `equal` para comparar os `vectors v1` e `v3`, que não são iguais.

Há outra versão da função `equal` que aceita uma função predicado binária como um quarto parâmetro. A função predicado binária recebe os dois elementos sob comparação e retorna um valor `bool` para indicar se os elementos são iguais. Isso pode ser útil em seqüências que armazenam objetos ou ponteiros para valores em vez de valores reais, porque você pode definir uma ou mais comparações. Por exemplo, você pode comparar a idade, o número do seguro social ou a localização de objetos `Employee` em vez de comparar objetos inteiros. Você pode comparar o que os ponteiros referenciam em vez de comparar os valores de ponteiro (isto é, os endereços armazenados nos ponteiros).

As linhas 39–43 começam instanciando um `pair` de iteradores chamado `location` para um `vector` de inteiros. Esse objeto armazena o resultado da chamada a `mismatch` (linha 43). A função `mismatch` compara duas seqüências de valores e retorna um `pair` de iteradores que indicam a localização em cada seqüência dos elementos não-correspondidos. Se houver correspondência de todos os elementos, os dois iteradores no `pair` são iguais ao último iterador de cada seqüência. Os três argumentos de iterador devem ser pelo menos iteradores de entrada. A linha 45 determina a localização real da não-correspondência nos `vectors` com a expressão `location.first - v1.begin()`. O resultado desse cálculo é o número de elementos entre os iteradores (isso é análogo à aritmética de ponteiro, que estudamos no Capítulo 8). Isso corresponde ao número do elemento nesse exemplo, porque a comparação é realizada desde o início de cada `vector`. Como com a função `equal`, há outra versão da função `mismatch` que aceita uma função predicado binária como um quarto parâmetro.

A linha 53 utiliza a função `lexicographical_compare` para comparar o conteúdo de dois arrays de caracteres. Os quatro argumentos de iterador dessa função devem ser pelo menos iteradores de entrada. Como você sabe, os ponteiros em arrays são iteradores de acesso aleatório. Os dois primeiros argumentos de iterador especificam o intervalo de localizações na primeira seqüência. Os dois últimos especificam o intervalo de localizações na segunda seqüência. Ao iterar pelas seqüências, o `lexicographical_compare` verifica se o elemento na primeira seqüência é menor que o elemento correspondente na segunda seqüência. Se for menor, a função retorna `true`. Se o elemento na primeira seqüência for maior que ou igual ao elemento na segunda seqüência, a função retorna `false`. Essa função pode ser utilizada para organizar seqüências lexicograficamente. Em geral, tais seqüências contêm strings.

```

1 // Figura 23.27: Fig23_27.cpp
2 // Funções equal, mismatch e lexicographical_compare da Standard Library.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <algorithm> // definições de algoritmo
8 #include <vector> // definição do template de classe vector
9 #include <iterator> // ostream_iterator
10
11 int main()
12 {
13 const int SIZE = 10;
14 int a1[SIZE] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
15 int a2[SIZE] = { 1, 2, 3, 4, 1000, 6, 7, 8, 9, 10 };
16 std::vector< int > v1(a1, a1 + SIZE); // cópia de a1
17 std::vector< int > v2(a1, a1 + SIZE); // cópia de a1
18 std::vector< int > v3(a2, a2 + SIZE); // cópia de a2
19 std::ostream_iterator< int > output(cout, " ");
20
21 cout << "Vector v1 contains: ";
22 std::copy(v1.begin(), v1.end(), output);
23 cout << "\nVector v2 contains: ";

```

**Figura 23.27** Algoritmos `equal`, `mismatch` e `lexicographical_compare`.

(continua)

```

24 std::copy(v2.begin(), v2.end(), output);
25 cout << "\nVector v3 contains: ";
26 std::copy(v3.begin(), v3.end(), output);
27
28 // compara a igualdade dos vetores v1 e v2
29 bool result = std::equal(v1.begin(), v1.end(), v2.begin());
30 cout << "\n\nVector v1 " << (result ? "is" : "is not")
31 << " equal to vector v2.\n";
32
33 // compara a igualdade dos vetores v1 e v3
34 result = std::equal(v1.begin(), v1.end(), v3.begin());
35 cout << "Vector v1 " << (result ? "is" : "is not")
36 << " equal to vector v3.\n";
37
38 // localização representa par de iteradores de vetor
39 std::pair< std::vector< int >::iterator,
40 std::vector< int >::iterator > location;
41
42 // verifica a não-correspondência entre v1 e v3
43 location = std::mismatch(v1.begin(), v1.end(), v3.begin());
44 cout << "\nThere is a mismatch between v1 and v3 at location "
45 << (location.first - v1.begin()) << "\nwhere v1 contains "
46 << *location.first << " and v3 contains " << *location.second
47 << "\n\n";
48
49 char c1[SIZE] = "HELLO";
50 char c2[SIZE] = "BYE BYE";
51
52 // realiza a comparação lexicográfica de c1 e c2
53 result = std::lexicographical_compare(c1, c1 + SIZE, c2, c2 + SIZE);
54 cout << c1 << (result ? " is less than " :
55 " is greater than or equal to ") << c2 << endl;
56 return 0;
57 } // fim de main

```

```

Vector v1 contains: 1 2 3 4 5 6 7 8 9 10
Vector v2 contains: 1 2 3 4 5 6 7 8 9 10
Vector v3 contains: 1 2 3 4 1000 6 7 8 9 10

```

```

Vector v1 is equal to vector v2.
Vector v1 is not equal to vector v3.

```

```

There is a mismatch between v1 and v3 at location 4
where v1 contains 5 and v3 contains 1000

```

```

HELLO is greater than or equal to BYE BYE

```

**Figura 23.27** Algoritmos `equal`, `mismatch` e `lexicographical_compare`.

(continuação)

### 23.5.3 `remove`, `remove_if`, `remove_copy` e `remove_copy_if`

A Figura 23.28 demonstra a remoção de valores de uma sequência com os algoritmos `remove`, `remove_if`, `remove_copy` e `remove_copy_if`.

A linha 26 utiliza a função `remove` para eliminar todos os elementos com o valor 10 no intervalo de `v.begin()` até, mas não incluindo, `v.end()` a partir de `v`. Os dois primeiros argumentos de iterador devem ser iteradores de leitura direta para que o algoritmo possa modificar os elementos na sequência. Essa função não modifica o número de elementos no vetor nem destrói os elementos eliminados, mas move todos os elementos que não são eliminados para o início do vetor. A função retorna um iterador posicionado depois do

último elemento de vector que não foi excluído. Os elementos desde a posição de iterador até o fim do vector têm valores indefinidos (nesse exemplo, cada posição ‘indefinida’ tem valor 0).

A linha 36 utiliza a função `remove_copy` para copiar todos os elementos que não têm o valor 10 no intervalo de `v2.begin()` até, mas não incluindo, `v2.end()` de `v2`. Os elementos são colocados em `c`, iniciando na posição `c.begin()`. Os iteradores fornecidos como os dois primeiros argumentos devem ser iteradores de entrada. O iterador fornecido como o terceiro argumento deve ser um iterador de saída para que o elemento sendo copiado possa ser inserido na localização de cópia. Essa função retorna um iterador posicionado depois do último elemento copiado em vector `c`. Note, na linha 31, o uso do construtor vector que recebe o número de elementos no vector e os valores iniciais desses elementos.

A linha 46 utiliza a função `remove_if` para excluir todos os elementos no intervalo de `v3.begin()` até, mas não incluindo, `v3.end()` de `v3` para o que nossa função predicado unária definida pelo usuário `greater9` retorne `true`. A função `greater9` (definida nas linhas 68–71) retorna `true` se o valor passado para ela for maior que 9; caso contrário, retorna `false`. Os iteradores fornecidos como os dois primeiros argumentos devem ser iteradores de leitura direta para que o algoritmo possa modificar os elementos na sequência. Essa função não modifica o número de elementos no vector, mas move para o início do vector todos os elementos que não são eliminados. Essa função retorna um iterador posicionado depois do último elemento no vector que não foi excluído. Todos os elementos da posição de iterador para o fim do vector têm valores indefinidos.

```

1 // Figura 23.28: Fig23_28.cpp
2 // Funções da Standard Library remove, remove_if,
3 // remove_copy e remove_copy_if.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include <algorithm> // definições de algoritmo
9 #include <vector> // definição do template de classe vector
10 #include <iterator> // ostream_iterator
11
12 bool greater9(int); // protótipo
13
14 int main()
15 {
16 const int SIZE = 10;
17 int a[SIZE] = { 10, 2, 10, 4, 16, 6, 14, 8, 12, 10 };
18 std::ostream_iterator< int > output(cout, " ");
19 std::vector< int > v(a, a + SIZE); // cópia de a
20 std::vector< int >::iterator newLastElement;
21
22 cout << "Vector v before removing all 10s:\n ";
23 std::copy(v.begin(), v.end(), output);
24
25 // remove todos os 10s de v
26 newLastElement = std::remove(v.begin(), v.end(), 10);
27 cout << "\nVector v after removing all 10s:\n ";
28 std::copy(v.begin(), newLastElement, output);
29
30 std::vector< int > v2(a, a + SIZE); // cópia de a
31 std::vector< int > c(SIZE, 0); // instancia vetor c
32 cout << "\n\nVector v2 before removing all 10s and copying:\n ";
33 std::copy(v2.begin(), v2.end(), output);
34
35 // copia de v2 para c, removendo 10s no processo
36 std::remove_copy(v2.begin(), v2.end(), c.begin(), 10);
37 cout << "\nVector c after removing all 10s from v2:\n ";
38 std::copy(c.begin(), c.end(), output);
39

```

**Figura 23.28** Os algoritmos `remove`, `remove_if`, `remove_copy` e `remove_copy_if`.

(continua)

```

40 std::vector< int > v3(a, a + SIZE); // cópia de a
41 cout << "\n\nVector v3 before removing all elements"
42 << "\ngreater than 9:\n ";
43 std::copy(v3.begin(), v3.end(), output);
44
45 // remove de v3 elementos maiores que 9
46 newLastElement = std::remove_if(v3.begin(), v3.end(), greater9);
47 cout << "\nVector v3 after removing all elements"
48 << "\ngreater than 9:\n ";
49 std::copy(v3.begin(), newLastElement, output);
50
51 std::vector< int > v4(a, a + SIZE); // cópia de a
52 std::vector< int > c2(SIZE, 0); // instancia vector c2
53 cout << "\n\nVector v4 before removing all elements"
54 << "\ngreater than 9 and copying:\n ";
55 std::copy(v4.begin(), v4.end(), output);
56
57 // copia elementos de v4 para c2, removendo elementos maiores
58 // que 9 no processo
59 std::remove_copy_if(v4.begin(), v4.end(), c2.begin(), greater9);
60 cout << "\nVector c2 after removing all elements"
61 << "\ngreater than 9 from v4:\n ";
62 std::copy(c2.begin(), c2.end(), output);
63 cout << endl;
64 return 0;
65 } // fim de main
66
67 // determina se o argumento é maior que 9
68 bool greater9(int x)
69 {
70 return x > 9;
71 } // fim da função greater9

```

Vector v before removing all 10s:

10 2 10 4 16 6 14 8 12 10

Vector v after removing all 10s:

2 4 16 6 14 8 12

Vector v2 before removing all 10s and copying:

10 2 10 4 16 6 14 8 12 10

Vector c after removing all 10s from v2:

2 4 16 6 14 8 12 0 0 0

Vector v3 before removing all elements  
greater than 9:

10 2 10 4 16 6 14 8 12 10

Vector v3 after removing all elements  
greater than 9:

2 4 6 8

Vector v4 before removing all elements  
greater than 9 and copying:

10 2 10 4 16 6 14 8 12 10

Vector c2 after removing all elements  
greater than 9 from v4:

2 4 6 8 0 0 0 0 0 0

**Figura 23.28** Os algoritmos `remove`, `remove_if`, `remove_copy` e `remove_copy_if`.

(continuação)

A linha 59 utiliza a função `remove_copy_if` para copiar todos esses elementos no intervalo de `v4.begin()` até, mas não incluindo, `v4.end()` a partir de `v4` para o qual a função predicado unária `greater9` retorna `true`. Os elementos são colocados em `c2`, iniciando na posição `c2.begin()`. Os iteradores fornecidos como os dois primeiros argumentos devem ser iteradores de entrada. O iterador fornecido como o terceiro argumento deve ser um iterador de saída para que o elemento sendo copiado possa ser inserido na localização de cópia. Essa função retorna um iterador posicionado depois do último elemento copiado para `c2`.

### 23.5.4 `replace`, `replace_if`, `replace_copy` e `replace_copy_if`

A Figura 23.29 demonstra a substituição de valores de uma sequência de algoritmos utilizando `replace`, `replace_if`, `replace_copy` e `replace_copy_if`.

A linha 25 utiliza a função `replace` para substituir todos os elementos com o valor 10 no intervalo de `v1.begin()` até, mas não incluindo, `v1.end()` em `v1` com o novo valor 100. Os iteradores fornecidos como os dois primeiros argumentos devem ser iteradores de leitura direta para que o algoritmo possa modificar os elementos na sequência.

A linha 35 utiliza a função `replace_copy` para copiar todos os elementos no intervalo de `v2.begin()` até, mas não incluindo, `v2.end()` de `v2`, substituindo todos os elementos com o valor 10 pelo novo valor 100. Os elementos são copiados em `c1`, iniciando na posição `c1.begin()`. Os iteradores fornecidos como os dois primeiros argumentos devem ser iteradores de entrada. O iterador fornecido como o terceiro argumento deve ser um iterador de saída para que o elemento sendo copiado possa ser inserido na localização da cópia. Essa função retorna um iterador posicionado depois do último elemento copiado para `c1`.

A linha 44 utiliza a função `replace_if` para substituir todos esses elementos no intervalo de `v3.begin()` até, mas não incluindo, `v3.end()` em `v3` para o qual a função predicado unária `greater9` retorna `true`. A função `greater9` (definida nas linhas 66–69) retorna

```

1 // Figura 23.29: Fig23_29.cpp
2 // Funções da Standard Library replace, replace_if,
3 // replace_copy e replace_copy_if.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include <algorithm>
9 #include <vector>
10 #include <iterator> // ostream_iterator
11
12 bool greater9(int); // protótipo de função predicado
13
14 int main()
15 {
16 const int SIZE = 10;
17 int a[SIZE] = { 10, 2, 10, 4, 16, 6, 14, 8, 12, 10 };
18 std::ostream_iterator< int > output(cout, " ");
19
20 std::vector< int > v1(a, a + SIZE); // cópia de a
21 cout << "Vector v1 before replacing all 10s:\n ";
22 std::copy(v1.begin(), v1.end(), output);
23
24 // substitui todos os 10s em v1 com 100
25 std::replace(v1.begin(), v1.end(), 10, 100);
26 cout << "\nVector v1 after replacing 10s with 100s:\n ";
27 std::copy(v1.begin(), v1.end(), output);
28
29 std::vector< int > v2(a, a + SIZE); // cópia de a
30 std::vector< int > c1(SIZE); // instancia vector c1
31 cout << "\n\nVector v2 before replacing all 10s and copying:\n ";
32 std::copy(v2.begin(), v2.end(), output);
33
34 // copia de v2 para c1, substituindo os 10s por 100s
35 std::replace_copy(v2.begin(), v2.end(), c1.begin(), 10, 100);

```

**Figura 23.29** Os algoritmos `replace`, `replace_if`, `replace_copy` e `replace_copy_if`.

(continua)

```

36 cout << "\nVector c1 after replacing all 10s in v2:\n ";
37 std::copy(c1.begin(), c1.end(), output);
38
39 std::vector< int > v3(a, a + SIZE); // cópia de a
40 cout << "\n\nVector v3 before replacing values greater than 9:\n ";
41 std::copy(v3.begin(), v3.end(), output);
42
43 // substitui valores maiores que 9 em v3 por 100
44 std::replace_if(v3.begin(), v3.end(), greater9, 100);
45 cout << "\nVector v3 after replacing all values greater"
46 << "\nthan 9 with 100s:\n ";
47 std::copy(v3.begin(), v3.end(), output);
48
49 std::vector< int > v4(a, a + SIZE); // cópia de a
50 std::vector< int > c2(SIZE); // instancia vetor c2'
51 cout << "\n\nVector v4 before replacing all values greater "
52 << "than 9 and copying:\n ";
53 std::copy(v4.begin(), v4.end(), output);
54
55 // copia v4 para c2, substituindo elementos maiores que 9 por 100
56 std::replace_copy_if(
57 v4.begin(), v4.end(), c2.begin(), greater9, 100);
58 cout << "\nVector c2 after replacing all values greater "
59 << "than 9 in v4:\n ";
60 std::copy(c2.begin(), c2.end(), output);
61 cout << endl;
62 return 0;
63 } // fim de main
64
65 // determina se o argumento é maior que 9
66 bool greater9(int x)
67 {
68 return x > 9;
69 } // fim da função greater9

```

Vector v1 before replacing all 10s:

10 2 10 4 16 6 14 8 12 10

Vector v1 after replacing 10s with 100s:

100 2 100 4 16 6 14 8 12 100

Vector v2 before replacing all 10s and copying:

10 2 10 4 16 6 14 8 12 10

Vector c1 after replacing all 10s in v2:

100 2 100 4 16 6 14 8 12 100

Vector v3 before replacing values greater than 9:

10 2 10 4 16 6 14 8 12 10

Vector v3 after replacing all values greater  
than 9 with 100s:

100 2 100 4 100 6 100 8 100 100

Vector v4 before replacing all values greater than 9 and copying:

10 2 10 4 16 6 14 8 12 10

Vector c2 after replacing all values greater than 9 in v4:

100 2 100 4 100 6 100 8 100 100

**Figura 23.29** Os algoritmos `replace`, `replace_if`, `replace_copy` e `replace_copy_if`.

(continuação)

true se o valor passado para ela for maior que 9; caso contrário, retorna false. O valor 100 substitui todos os valores maiores que 9. Os iteradores fornecidos como os dois primeiros argumentos devem ser iteradores de leitura direta para que o algoritmo possa modificar os elementos na sequência.

As linhas 56–57 utilizam a função `replace_copy_if` para copiar todos os elementos no intervalo de `v4.begin()` até, mas não incluindo, `v4.end()` a partir de `v4`. Os elementos pelos quais a função predicado unária `greater9` retorna true são substituídos pelo valor 100. Os elementos são colocados em `c2`, iniciando na posição `c2.begin()`. Os iteradores fornecidos como os dois primeiros argumentos devem ser iteradores de entrada. O iterador fornecido como o terceiro argumento deve ser um iterador de saída para que o elemento sendo copiado possa ser inserido na localização de cópia. Essa função retorna um iterador posicionado depois do último elemento copiado para `c2`.

### 23.5.5 Algoritmos matemáticos

A Figura 23.30 demonstra vários algoritmos matemáticos comuns da STL, incluindo `random_shuffle`, `count`, `count_if`, `min_element`, `max_element`, `accumulate`, `for_each` e `transform`.

A linha 26 utiliza a função `random_shuffle` para reordenar aleatoriamente os elementos no intervalo de `v.begin()` até, mas não incluindo, `v.end()` em `v`. Essa função aceita dois argumentos de iterador de acesso aleatório.

A linha 36 utiliza a função `count` para contar os elementos com o valor 8 no intervalo de `v2.begin()` até, mas não incluindo, `v2.end()` em `v2`. Essa função requer que seus dois argumentos de iterador sejam pelo menos iteradores de entrada.

A linha 40 utiliza a função `count_if` para contar os elementos no intervalo de `v2.begin()` até, mas não incluindo, `v2.end()` em `v2` para o que a função predicado `greater9` retorna true. A função `count_if` requer que seus dois argumentos de iterador sejam pelo menos iteradores de entrada.

A linha 45 utiliza a função `min_element` para localizar o menor elemento no intervalo de `v2.begin()` até, mas não incluindo, `v2.end()` em `v2`. A função retorna um iterador de leitura direta localizado no menor elemento ou, se o intervalo estiver vazio, retorna `v2.end()`. A função requer que seus dois argumentos de iterador sejam pelo menos iteradores de entrada. Uma segunda versão dessa função aceita como seu terceiro argumento uma função binária que compara os elementos na sequência. A função binária aceita dois argumentos e retorna um valor bool.

```

1 // Figura 23.30: Fig23_30.cpp
2 // Algoritmos matemáticos da Standard Library.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <algorithm> // definições de algoritmo
8 #include <numeric> // accumulate é definida aqui
9 #include <vector>
10 #include <iterator>
11
12 bool greater9(int); // protótipo de função predicado
13 void outputSquare(int); // gera saída do quadrado de um valor
14 int calculateCube(int); // calcula o cubo de um valor
15
16 int main()
17 {
18 const int SIZE = 10;
19 int a1[SIZE] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
20 std::vector< int > v(a1, a1 + SIZE); // cópia de a1
21 std::ostream_iterator< int > output(cout, " ");
22
23 cout << "Vector v before random_shuffle: ";
24 std::copy(v.begin(), v.end(), output);
25
26 std::random_shuffle(v.begin(), v.end()); // embaralha os elementos de v
27 cout << "\nVector v after random_shuffle: ";
28 std::copy(v.begin(), v.end(), output);
29

```

Figura 23.30 Algoritmos matemáticos da Standard Library.

(continua)



```

30 int a2[SIZE] = { 100, 2, 8, 1, 50, 3, 8, 8, 9, 10 };
31 std::vector< int > v2(a2, a2 + SIZE); // cópia de a2
32 cout << "\n\nVector v2 contains: ";
33 std::copy(v2.begin(), v2.end(), output);
34
35 // conta o número de elementos em v2 com valor 8
36 int result = std::count(v2.begin(), v2.end(), 8);
37 cout << "\nNumber of elements matching 8: " << result;
38
39 // conta o número de elementos em v2 que são maiores que 9
40 result = std::count_if(v2.begin(), v2.end(), greater9);
41 cout << "\nNumber of elements greater than 9: " << result;
42
43 // localiza o elemento mínimo em v2
44 cout << "\n\nMinimum element in Vector v2 is: "
45 << *(std::min_element(v2.begin(), v2.end()));
46
47 // localiza o elemento máximo em v2
48 cout << "\nMaximum element in Vector v2 is: "
49 << *(std::max_element(v2.begin(), v2.end()));
50
51 // calcula a soma de elementos em v
52 cout << "\n\nThe total of the elements in Vector v is: "
53 << std::accumulate(v.begin(), v.end(), 0);
54
55 // gera saída de quadrado de cada elemento em v
56 cout << "\n\nThe square of every integer in Vector v is:\n";
57 std::for_each(v.begin(), v.end(), outputSquare);
58
59 std::vector< int > cubes(SIZE); // instancia vector cubes
60
61 // calcula o cubo de cada elemento em v; coloca resultados em cubes
62 std::transform(v.begin(), v.end(), cubes.begin(), calculateCube);
63 cout << "\n\nThe cube of every integer in Vector v is:\n";
64 std::copy(cubes.begin(), cubes.end(), output);
65 cout << endl;
66 return 0;
67 } // fim de main
68
69 // determina se o argumento é maior que 9
70 bool greater9(int value)
71 {
72 return value > 9;
73 } // fim da função greater9
74
75 // gera saída do quadrado do argumento
76 void outputSquare(int value)
77 {
78 cout << value * value << ' ';
79 } // fim da função outputSquare
80
81 // retorna o cubo do argumento
82 int calculateCube(int value)
83 {
84 return value * value * value;
85 } // fim da função calculateCube

```

Figura 23.30 Algoritmos matemáticos da Standard Library.

(continua)

```

Vector v before random_shuffle: 1 2 3 4 5 6 7 8 9 10
Vector v after random_shuffle: 5 4 1 3 7 8 9 10 6 2

Vector v2 contains: 100 2 8 1 50 3 8 8 9 10
Number of elements matching 8: 3
Number of elements greater than 9: 3

Minimum element in Vector v2 is: 1
Maximum element in Vector v2 is: 100

The total of the elements in Vector v is: 55

The square of every integer in Vector v is:
25 16 1 9 49 64 81 100 36 4

The cube of every integer in Vector v is:
125 64 1 27 343 512 729 1000 216 8

```

Figura 23.30 Algoritmos matemáticos da Standard Library.

(continuação)



## Boa prática de programação 23.2

*É uma boa prática verificar se o intervalo especificado em uma chamada para `min_element` não está vazio e se o valor de retorno não é o iterador ‘além do fim’.*

A linha 49 utiliza a função `max_element` para localizar o maior elemento no intervalo de `v2.begin()` até, mas não incluindo, `v2.end()` em `v2`. A função retorna um iterador de entrada localizado no maior elemento. A função requer que seus dois argumentos de iterador sejam pelo menos iteradores de entrada. Uma segunda versão dessa função aceita como seu terceiro argumento uma função predicado binária que compara os elementos na sequência. A função binária aceita dois argumentos e retorna um valor `bool`.

A linha 53 utiliza a função `accumulate` (cujo template está no arquivo de cabeçalho `<numeric>`) para somar os valores no intervalo de `v.begin()` até, mas não incluindo, `v.end()` em `v`. Os dois argumentos de iterador da função devem ser pelo menos iteradores de entrada e seu terceiro argumento representa o valor inicial do total. Uma segunda versão dessa função aceita como seu quarto argumento uma função geral que determina como os elementos são acumulados. A função geral deve aceitar dois argumentos e retornar um resultado. O primeiro argumento para essa função é o valor atual da acumulação. O segundo argumento é o valor do elemento atual na sequência sendo acumulada.

A linha 57 utiliza a função `for_each` para aplicar uma função geral a cada elemento no intervalo de `v.begin()` até, mas não incluindo, `v.end()` em `v`. A função geral deve aceitar o elemento atual como um argumento e não deve modificar esse elemento. A função `for_each` requer que seus dois argumentos de iterador sejam pelo menos iteradores de entrada.

A linha 62 utiliza a função `transform` para aplicar uma função geral a cada elemento no intervalo de `v.begin()` até, mas não incluindo, `v.end()` em `v`. A função geral (o quarto argumento) deve aceitar o elemento atual como um argumento, não deve modificar o elemento e deve retornar o valor transformado (`transformed`). A função `transform` requer que seus dois primeiros argumentos de iterador sejam pelo menos iteradores de entrada e que seu terceiro argumento seja pelo menos um iterador de saída. O terceiro argumento especifica onde os valores transformados devem ser colocados. Observe que o terceiro argumento pode ser igual ao primeiro. Outra versão de `transform` aceita cinco argumentos — os dois primeiros argumentos são iteradores de entrada que especificam um intervalo de elementos de um contêiner de origem, o terceiro argumento é um iterador de entrada que especifica o primeiro elemento em outro contêiner de origem, o quarto argumento é um iterador de saída que especifica onde os valores transformados devem ser colocados e o último argumento é uma função geral que aceita dois argumentos. Essa versão de `transform` pega um elemento de cada uma das duas origens de entrada e aplica a função geral a esse par de elementos e, então, coloca o valor transformado na localização especificada pelo quarto argumento.

### 23.5.6 Algoritmos de pesquisa e classificação básica

A Figura 23.31 demonstra algumas capacidades básicas de pesquisa e classificação da Standard Library, incluindo `find`, `find_if`, `sort` e `binary_search`.

A linha 25 utiliza a função `find` para localizar o valor 16 no intervalo de `v.begin()` até, mas não incluindo, `v.end()` em `v`. A função requer que seus dois argumentos de iterador sejam pelo menos iteradores de entrada e retorna um iterador de entrada que está posicionado no primeiro elemento contendo o valor ou que indica o fim da sequência (como é o caso na linha 33).

```

1 // Figura 23.31: Fig23_31.cpp
2 // Algoritmos de pesquisa e classificação da Standard Library.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <algorithm> // definições de algoritmo
8 #include <vector> // definição do template de classe vector
9 #include <iterator>
10
11 bool greater10(int value); // protótipo de função predicado
12
13 int main()
14 {
15 const int SIZE = 10;
16 int a[SIZE] = { 10, 2, 17, 5, 16, 8, 13, 11, 20, 7 };
17 std::vector< int > v(a, a + SIZE); // cópia de a
18 std::ostream_iterator< int > output(cout, " ");
19
20 cout << "Vector v contains: ";
21 std::copy(v.begin(), v.end(), output); // exibe vetor de saída
22
23 // localiza primeira ocorrência de 16 em v
24 std::vector< int >::iterator location;
25 location = std::find(v.begin(), v.end(), 16);
26
27 if (location != v.end()) // localizou 16
28 cout << "\n\nFound 16 at location " << (location - v.begin());
29 else // 16 não localizado
30 cout << "\n\n16 not found";
31
32 // localiza primeira ocorrência de 100 em v
33 location = std::find(v.begin(), v.end(), 100);
34
35 if (location != v.end()) // localizou 100
36 cout << "\n\nFound 100 at location " << (location - v.begin());
37 else // 100 não localizado
38 cout << "\n\n100 not found";
39
40 // localiza primeira ocorrência de valor maior que 10 em v
41 location = std::find_if(v.begin(), v.end(), greater10);
42
43 if (location != v.end()) // localizou valor maior que 10
44 cout << "\n\nThe first value greater than 10 is " << *location
45 << "\n\nfound at location " << (location - v.begin());
46 else // valor maior que 10 não localizado
47 cout << "\n\nNo values greater than 10 were found";
48
49 // classifica elementos de v
50 std::sort(v.begin(), v.end());
51 cout << "\n\nVector v after sort: ";
52 std::copy(v.begin(), v.end(), output);
53
54 // usa binary_search para localizar 13 em v
55 if (std::binary_search(v.begin(), v.end(), 13))
56 cout << "\n\n13 was found in v";

```

**Figura 23.31** Algoritmos de pesquisa e classificação básicos da Standard Library.

(continua)

```

57 else
58 cout << "\n13 was not found in v";
59
60 // usa binary_search para localizar 100 em v
61 if (std::binary_search(v.begin(), v.end(), 100))
62 cout << "\n100 was found in v";
63 else
64 cout << "\n100 was not found in v";
65
66 cout << endl;
67 return 0;
68 } // fim de main
69
70 // determina se o argumento é maior que 10
71 bool greater10(int value)
72 {
73 return value > 10;
74 } // fim da função greater10

```

Vector v contains: 10 2 17 5 16 8 13 11 20 7

Found 16 at location 4  
100 not found

The first value greater than 10 is 17  
found at location 2

Vector v after sort: 2 5 7 8 10 11 13 16 17 20

13 was found in v  
100 was not found in v

**Figura 23.31** Algoritmos de pesquisa e classificação básicos da Standard Library.

(continuação)

A linha 41 utiliza a função **find\_if** para localizar o primeiro valor no intervalo de `v.begin()` até, mas não incluindo, `v.end()` em `v` para o qual a função predicado unária `greater10` retorna `true`. A função `greater10` (definida nas linhas 71–74) aceita um inteiro e retorna um valor `bool` que indica se o argumento de inteiro é maior que 10. A função `find_if` requer que seus dois argumentos de iterador sejam pelo menos iteradores de entrada. A função retorna um iterador de entrada que é posicionado no primeiro elemento contendo um valor para o qual a função predicado retorna `true` ou indica o fim da sequência.

A linha 50 utiliza a função **sort** para organizar os elementos no intervalo de `v.begin()` até, mas não incluindo, `v.end()` em `v` na ordem crescente. A função requer que seus dois argumentos de iterador sejam iteradores de acesso aleatório. Uma segunda versão dessa função aceita um terceiro argumento que é uma função predicado binária aceitando dois argumentos que são valores na sequência e retornando um `bool` que indica a ordem de classificação — se o valor de retorno for `true`, os dois elementos sendo comparados estarão em ordem de classificação.



### Erro comum de programação 23.5

*Tentar classificar (sort) um contêiner utilizando um iterador diferente de um iterador de acesso aleatório é um erro de compilação. A função sort requer um iterador de acesso aleatório.*

A linha 55 utiliza a função **binary\_search** para determinar se o valor 13 está no intervalo de `v.begin()` até, mas não incluindo, `v.end()` em `v`. A sequência de valores deve ser classificada em ordem crescente primeiro. A função `binary_search` requer que seus dois argumentos de iterador sejam pelo menos iteradores de leitura direta. A função retorna um `bool` que indica se o valor foi localizado na sequência. A linha 61 demonstra uma chamada para a função `binary_search` em que o valor não é encontrado. Uma segunda versão dessa função aceita um quarto argumento que é uma função predicado binária aceitando dois argumentos que são valores na sequência e retornando um `bool`. A função predicado retorna `true` se os dois elementos comparados estiverem na ordem de classificação.

### 23.5.7 swap, iter\_swap e swap\_ranges

A Figura 23.32 demonstra os algoritmos `swap`, `iter_swap` e `swap_ranges` para permutar elementos. A linha 20 utiliza a função `swap` para permutar dois valores. Nesse exemplo, o primeiro e o segundo elementos do array `a` são permutados. A função aceita como argumentos referências a dois valores sendo permutados.

```

1 // Figura 23.32: Fig23_32.cpp
2 // Algoritmos iter_swap, swap e swap_ranges da Standard Library.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <algorithm> // definições de algoritmo
8 #include <iterator>
9
10 int main()
11 {
12 const int SIZE = 10;
13 int a[SIZE] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
14 std::ostream_iterator< int > output(cout, " ");
15
16 cout << "Array a contains:\n ";
17 std::copy(a, a + SIZE, output); // exibe array a
18
19 // permuta elementos nas localizações 0 e 1 do array a
20 std::swap(a[0], a[1]);
21
22 cout << "\nArray a after swapping a[0] and a[1] using swap:\n ";
23 std::copy(a, a + SIZE, output); // exibe array a
24
25 // utiliza iteradores para permutar elementos nas localizações 0 e 1 do array a
26 std::iter_swap(&a[0], &a[1]); // permuta com iteradores
27 cout << "\nArray a after swapping a[0] and a[1] using iter_swap:\n ";
28 std::copy(a, a + SIZE, output);
29
30 // permuta elementos nos primeiros cinco elementos do array a pelos
31 // elementos nos últimos cinco elementos do array a
32 std::swap_ranges(a, a + 5, a + 5);
33
34 cout << "\nArray a after swapping the first five elements\n"
35 << "with the last five elements:\n ";
36 std::copy(a, a + SIZE, output);
37 cout << endl;
38 return 0;
39 } // fim de main

```

```

Array a contains:
 1 2 3 4 5 6 7 8 9 10
Array a after swapping a[0] and a[1] using swap:
 2 1 3 4 5 6 7 8 9 10
Array a after swapping a[0] and a[1] using iter_swap:
 2 1 3 4 5 6 7 8 9 10
Array a after swapping the first five elements
with the last five elements:
 6 7 8 9 10 1 2 3 4 5

```

**Figura 23.32** Demonstrando `swap`, `iter_swap` e `swap_ranges`.

A linha 26 utiliza a função `iter_swap` para permutar os dois elementos. A função aceita dois argumentos de iterador de leitura direta (nesse caso, ponteiros para elementos do array `a`) e permuta os valores nos elementos referenciados pelos iteradores.

A linha 32 utiliza a função `swap_ranges` para permutar os elementos no intervalo de `a` até, mas não incluindo, `a + 5` pelos elementos que iniciam na posição `a + 5`. A função requer três argumentos de iterador de leitura direta. Os dois primeiros argumentos especificam o intervalo de elementos na primeira sequência que serão permutados pelos elementos na segunda sequência iniciando do iterador no terceiro argumento. Nesse exemplo, as duas sequências de valores estão no mesmo array, mas as sequências podem ser de arrays ou contêineres diferentes.

### 23.5.8 `copy_backward`, `merge`, `unique` e `reverse`

A Figura 23.33 demonstra os algoritmos STL `copy_backward`, `merge`, `unique` e `reverse`. A linha 28 utiliza a função `copy_backward` para copiar elementos no intervalo de `v1.begin()` até, mas não incluindo, `v1.end()` em `v1`, colocando os elementos em `results` iniciando do elemento antes de `results.end()` e direcionando para o início do vector. A função retorna um iterador posicionado no último elemento copiado para o `results` (por exemplo, o início de `results`, por causa da cópia de trás para frente). Os elementos são colocados em `results` na mesma ordem de `v1`. Essa função requer três argumentos de iterador bidirecional (iteradores que podem ser incrementados e decrementados para iterar para frente e para trás por uma sequência, respectivamente). A principal diferença entre `copy` e `copy_backward` é que o iterador retornado de `copy` é posicionado *depois* do último elemento copiado e o iterador retornado de `copy_backward` é posicionado *no* último elemento copiado (que na realidade é o primeiro elemento na sequência). Além disso, `copy` requer dois iteradores de entrada e um iterador de saída como argumento.

As linhas 35–36 utilizam a função `merge` para combinar duas sequências crescentes de valores classificadas em uma terceira sequência crescente classificada. A função requer cinco argumentos de iterador. Os primeiros quatro devem ser pelo menos iteradores de entrada e o último deve ser pelo menos um iterador de saída. Os dois primeiros argumentos especificam o intervalo de elementos na primeira sequência classificada (`v1`), os dois segundos argumentos especificam o intervalo de elementos na segunda sequência classificada (`v2`) e o último argumento especifica a localização inicial na terceira sequência (`results2`) em que os elementos serão mesclados. Uma segunda versão dessa função aceita como seu sexto argumento uma função predicado binária que especifica a ordem de classificação.

Observe que a linha 32 cria o vetor `results2` com o número de elementos `v1.size() + v2.size()`. Utilizar a função `merge` como mostrado aqui requer que a sequência em que os resultados são armazenados tenha pelo menos o tamanho das duas sequências sendo mescladas. Se você não quiser alocar o número de elementos para a sequência resultante antes da operação `merge`, pode utilizar as seguintes instruções:

```
std::vector< int > results2();
std::merge (v1.begin(), v1.end(), v2.begin(), v2.end(),
 std::back_inserter(results2));
```

O argumento `std::back_inserter( results2 )` utiliza o template de função `back_inserter` (arquivo de cabeçalho `<iterator>`) para o contêiner `results2`. Um `back_inserter` chama a função `push_back` padrão do contêiner para inserir um elemento no fim do contêiner. Mais importante, se um elemento é inserido em um contêiner que não tem mais nenhum espaço disponível, o tamanho do contêiner cresce. Portanto, não é preciso saber antecipadamente o número de elementos do contêiner. Há dois outros inserters — `front_inserter` (para inserir um elemento no início de um contêiner especificado como seu argumento) e `inserter` (para inserir um elemento antes do iterador fornecido como seu segundo argumento no contêiner fornecido como seu primeiro argumento).

A linha 43 utiliza a função `unique` na sequência de ordenação de elementos no intervalo de `results2.begin()` até, mas não incluindo, `results2.end()` em `results2`. Depois que essa função é aplicada a uma sequência classificada com valores duplicados, somente uma única cópia de cada valor permanece na sequência. A função aceita dois argumentos que devem ser pelo menos iteradores de leitura direta. A função retorna um iterador posicionado depois do último elemento na sequência de valores únicos. Os valores de todos os elementos no contêiner depois do último valor único são indefinidos. Uma segunda versão dessa função aceita como um terceiro argumento uma função predicado binária que especifica como comparar a igualdade de dois elementos.

```
1 // Figura 23.33: Fig23_33.cpp
2 // Funções copy_backward, merge, unique e reverse da Standard Library.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <algorithm> // definições de algoritmo
8 #include <vector> // definição do template de classe vector
9 #include <iterator> // ostream_iterator
10
```

**Figura 23.33** Demonstrando `copy_backward`, `merge`, `unique` e `reverse`.

(continua)

```

11 int main()
12 {
13 const int SIZE = 5;
14 int a1[SIZE] = { 1, 3, 5, 7, 9 };
15 int a2[SIZE] = { 2, 4, 5, 7, 9 };
16 std::vector< int > v1(a1, a1 + SIZE); // cópia de a1
17 std::vector< int > v2(a2, a2 + SIZE); // cópia de a2
18 std::ostream_iterator< int > output(cout, " ");
19
20 cout << "Vector v1 contains: ";
21 std::copy(v1.begin(), v1.end(), output); // exibe saída de vector
22 cout << "\nVector v2 contains: ";
23 std::copy(v2.begin(), v2.end(), output); // exibe saída de vector
24
25 std::vector< int > results(v1.size());
26
27 // coloca elementos de v1 em results na ordem inversa
28 std::copy_backward(v1.begin(), v1.end(), results.end());
29 cout << "\n\nAfter copy_backward, results contains: ";
30 std::copy(results.begin(), results.end(), output);
31
32 std::vector< int > results2(v1.size() + v2.size());
33
34 // mescla elementos de v1 e v2 em results2 na ordem classificada
35 std::merge(v1.begin(), v1.end(), v2.begin(), v2.end(),
36 results2.begin());
37
38 cout << "\n\nAfter merge of v1 and v2 results2 contains:\n";
39 std::copy(results2.begin(), results2.end(), output);
40
41 // elimina valores duplicados de results2
42 std::vector< int >::iterator endLocation;
43 endLocation = std::unique(results2.begin(), results2.end());
44
45 cout << "\n\nAfter unique results2 contains:\n";
46 std::copy(results2.begin(), endLocation, output);
47
48 cout << "\n\nVector v1 after reverse: ";
49 std::reverse(v1.begin(), v1.end()); // inverte elementos de v1
50 std::copy(v1.begin(), v1.end(), output);
51 cout << endl;
52 return 0;
53 } // fim de main

```

Vector v1 contains: 1 3 5 7 9  
 Vector v2 contains: 2 4 5 7 9

After copy\_backward, results contains: 1 3 5 7 9

After merge of v1 and v2 results2 contains:  
 1 2 3 4 5 5 7 7 9 9

After unique results2 contains:  
 1 2 3 4 5 7 9

Vector v1 after reverse: 9 7 5 3 1

**Figura 23.33** Demonstrando copy\_backward, merge, unique e reverse.

(continuação)



A linha 49 utiliza a função `reverse` para inverter todos os elementos no intervalo de `v1.begin()` até, mas não incluindo, `v1.end()` em `v1`. A função aceita dois argumentos que devem ser iteradores pelo menos bidirecionais.

### 23.5.9 `inplace_merge`, `unique_copy` e `reverse_copy`

A Figura 23.34 demonstra os algoritmos STL `inplace_merge`, `unique_copy` e `reverse_copy`. A linha 24 utiliza a função `inplace_merge` para mesclar duas seqüências classificadas de elementos no mesmo contêiner. Nesse exemplo, os elementos a partir de `v1.begin()` até, mas não incluindo, `v1.begin() + 5` são mesclados com os elementos de `v1.begin() + 5` até, mas não incluindo, `v1.end()`. Essa função requer seus três argumentos de iterador sejam pelo menos iteradores bidirecionais. Uma segunda versão dessa função aceita como um quarto argumento uma função predicado binária para comparar elementos nas duas seqüências.

As linhas 32–33 utilizam a função `unique_copy` para fazer uma cópia de todos os elementos únicos na seqüência classificada de valores de `v1.begin()` até, mas não incluindo, `v1.end()`. Os elementos copiados são colocados no vector `results1`. Os dois primeiros argumentos devem ser pelo menos iteradores de entrada e o último deve ser pelo menos um iterador de saída. Nesse exemplo, não pré-alocamos elementos suficientes em `results1` para armazenar todos os elementos copiados de `v1`. Em vez disso, utilizamos a função `back_inserter` (definida no arquivo de cabeçalho `<iterator>`) para adicionar elementos ao fim de `v1`. A função `back_inserter` utiliza a capacidade da classe `vector` de inserir elementos no fim do vector. Como o `back_inserter` insere um elemento em vez de substituir o valor de um elemento existente, o vector é capaz de crescer para acomodar elementos adicionais. Uma segunda versão da função `unique_copy` aceita como um quarto argumento uma função predicado binária para comparar a igualdade de elementos.

As linhas 40–41 utilizam a função `reverse_copy` para fazer uma cópia invertida dos elementos no intervalo de `v1.begin()` até, mas não incluindo, `v1.end()`. Os elementos copiados são inseridos em `results2` utilizando um objeto `back_inserter` para assegurar que o vector possa crescer para acomodar o número apropriado de elementos copiados. A função `reverse_copy` requer que seus dois primeiros argumentos de iterador sejam pelo menos iteradores bidirecionais e que seu terceiro seja pelo menos um iterador de saída.

### 23.5.10 Operações set

A Figura 23.35 demonstra as funções Standard Library `includes`, `set_difference`, `set_intersection`, `set_symmetric_difference` e `set_union` para manipular conjuntos de valores classificados. Para demonstrar que as funções Standard Library podem ser aplicadas a arrays e contêineres, esse exemplo utiliza somente arrays (lembre-se, um ponteiro em um array é um iterador de acesso aleatório).

As linhas 27 e 33 chamam a função `includes` nas condições de instruções `if`. A função `includes` compara dois conjuntos de valores classificados para determinar se cada elemento do segundo conjunto está ou não no primeiro conjunto. Se estiver, `includes` retorna `true`; caso contrário, retorna `false`. Os dois primeiros argumentos de iterador devem ser pelo menos iteradores de entrada e descrever o primeiro conjunto de valores. Na linha 27, o primeiro conjunto consiste nos elementos de `a1` até, mas não incluindo, `a1 + SIZE1`. Os últimos dois argumentos de iterador devem ser pelo menos iteradores de entrada e devem descrever o segundo conjunto de valores. Nesse exemplo, o segundo conjunto consiste nos elementos de `a2` até, mas não incluindo, `a2 + SIZE2`. Uma segunda versão da função `includes` aceita um quinto argumento que é uma função predicado binária para comparar a igualdade de elementos.

As linhas 41–42 utilizam a função `set_difference` para localizar os elementos do primeiro conjunto de valores classificados que não estão no segundo conjunto de valores classificados (ambos os conjuntos de valores devem estar na ordem crescente). Os elementos que são diferentes são copiados no quinto argumento (nesse caso, o array `difference`). Os dois primeiros argumentos de iterador devem ser pelo menos iteradores de entrada para o primeiro conjunto de valores. Os dois próximos argumentos de iterador devem ser pelo menos iteradores de entrada para o segundo conjunto de valores. O quinto argumento deve ser pelo menos um iterador de saída que indica onde armazenar uma cópia dos valores que são diferentes. A função retorna um iterador de saída posicionado imediatamente depois do último valor copiado no conjunto para o qual o quinto argumento aponta. Uma segunda versão de função `set_difference` aceita um sexto argumento que é uma função predicado binária para indicar a ordem em que os elementos foram originalmente classificados. As duas seqüências devem ser classificadas utilizando a mesma função de comparação.

As linhas 49–50 utilizam a função `set_intersection` para determinar os elementos do primeiro conjunto de valores classificados que estão no segundo conjunto de valores classificados (ambos os conjuntos de valores devem estar na ordem crescente). Os elementos comuns a ambos os conjuntos são copiados para o quinto argumento (nesse caso, o array `intersection`). Os dois primeiros argumentos de iterador devem ser pelo menos iteradores de entrada para o primeiro conjunto de valores. Os dois próximos argumentos de iterador devem ser pelo menos iteradores de entrada para o segundo conjunto de valores. O quinto argumento deve ser pelo menos um iterador de saída que indica onde armazenar uma cópia dos valores que são os mesmos. A função retorna um iterador de saída posicionado imediatamente depois do último valor copiado no conjunto para o qual o quinto argumento aponta. Uma segunda versão de função `set_intersection` aceita um sexto argumento que é uma função predicado binária para indicar a ordem em que os elementos foram originalmente classificados. As duas seqüências devem ser classificadas utilizando a mesma função de comparação.

As linhas 58–59 utilizam a função `set_symmetric_difference` para determinar os elementos no primeiro conjunto que não estão no segundo conjunto e os elementos no segundo conjunto que não estão no primeiro (ambos os conjuntos devem estar na ordem crescente). Os elementos que são diferentes são copiados a partir de ambos os conjuntos para o quinto argumento (o array `symmetric_difference`). Os dois primeiros argumentos de iterador devem ser pelo menos iteradores de entrada para o primeiro conjunto de valores. Os dois próximos argumentos de iterador devem ser pelo menos iteradores de entrada para o segundo conjunto de valores. O quinto argumento deve ser pelo menos um iterador de saída que indica onde armazenar uma cópia dos valores que são diferentes. A função retorna um iterador de saída posicionado imediatamente depois do último valor copiado no conjunto para o qual o quinto argumento aponta. Uma segunda versão de função `set_symmetric_difference` aceita um sexto argumento que é uma função predicado binária para indicar a

```

1 // Figura 23.34: Fig23_34.cpp
2 // Algoritmos da Standard Library inplace_merge,
3 // reverse_copy e unique_copy.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include <algorithm> // definições de algoritmo
9 #include <vector> // definição do template de classe vector
10 #include <iterator> // definição back_inserter
11
12 int main()
13 {
14 const int SIZE = 10;
15 int a1[SIZE] = { 1, 3, 5, 7, 9, 1, 3, 5, 7, 9 };
16 std::vector< int > v1(a1, a1 + SIZE); // cópia de a
17 std::ostream_iterator< int > output(cout, " ");
18
19 cout << "Vector v1 contains: ";
20 std::copy(v1.begin(), v1.end(), output);
21
22 // mescla primeira metade de v1 com a segunda metade de v1 de modo que
23 // v1 contenha o conjunto classificado de elementos depois da mesclagem
24 std::inplace_merge(v1.begin(), v1.begin() + 5, v1.end());
25
26 cout << "\nAfter inplace_merge, v1 contains: ";
27 std::copy(v1.begin(), v1.end(), output);
28
29 std::vector< int > results1;
30
31 // copia somente elementos únicos de v1 para results1
32 std::unique_copy(
33 v1.begin(), v1.end(), std::back_inserter(results1));
34 cout << "\nAfter unique_copy results1 contains: ";
35 std::copy(results1.begin(), results1.end(), output);
36
37 std::vector< int > results2;
38
39 // copia elementos de v1 para results2 na ordem inversa
40 std::reverse_copy(
41 v1.begin(), v1.end(), std::back_inserter(results2));
42 cout << "\nAfter reverse_copy, results2 contains: ";
43 std::copy(results2.begin(), results2.end(), output);
44 cout << endl;
45 return 0;
46 } // fim de main

```

Vector v1 contains: 1 3 5 7 9 1 3 5 7 9  
 After inplace\_merge, v1 contains: 1 1 3 3 5 5 7 7 9 9  
 After unique\_copy results1 contains: 1 3 5 7 9  
 After reverse\_copy, results2 contains: 9 9 7 7 5 5 3 3 1 1

**Figura 23.34** Demonstrando inplace\_merge, unique\_copy e reverse\_copy.

```

1 // Figura 23.35: Fig23_35.cpp
2 // Algoritmos includes, set_difference,
3 // set_intersection, set_symmetric_difference e set_union da Standard Library.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include <algorithm> // definições de algoritmo
9 #include <iterator> // ostream_iterator
10
11 int main()
12 {
13 const int SIZE1 = 10, SIZE2 = 5, SIZE3 = 20;
14 int a1[SIZE1] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
15 int a2[SIZE2] = { 4, 5, 6, 7, 8 };
16 int a3[SIZE2] = { 4, 5, 6, 11, 15 };
17 std::ostream_iterator< int > output(cout, " ");
18
19 cout << "a1 contains: ";
20 std::copy(a1, a1 + SIZE1, output); // exibe o array a1
21 cout << "\na2 contains: ";
22 std::copy(a2, a2 + SIZE2, output); // exibe o array a2
23 cout << "\na3 contains: ";
24 std::copy(a3, a3 + SIZE2, output); // exibe o array a3
25
26 // determina se o conjunto a2 está completamente contido em a1
27 if (std::includes(a1, a1 + SIZE1, a2, a2 + SIZE2))
28 cout << "\na1 includes a2";
29 else
30 cout << "\na1 does not include a2";
31
32 // determina se o conjunto a3 está completamente contido em a1
33 if (std::includes(a1, a1 + SIZE1, a3, a3 + SIZE2))
34 cout << "\na1 includes a3";
35 else
36 cout << "\na1 does not include a3";
37
38 int difference[SIZE1];
39
40 // determina elementos de a1 não em a2
41 int *ptr = std::set_difference(a1, a1 + SIZE1,
42 a2, a2 + SIZE2, difference);
43 cout << "\nset_difference of a1 and a2 is: ";
44 std::copy(difference, ptr, output);
45
46 int intersection[SIZE1];
47
48 // determina elementos tanto em a1 como em a2
49 ptr = std::set_intersection(a1, a1 + SIZE1,
50 a2, a2 + SIZE2, intersection);
51 cout << "\nset_intersection of a1 and a2 is: ";
52 std::copy(intersection, ptr, output);
53
54 int symmetric_difference[SIZE1 + SIZE2];
55

```

Figura 23.35 Operações set da Standard Library.

(continua)

```

56 // determina elementos de a1 que não estão em a2 e
57 // elementos de a2 que não estão em a1
58 ptr = std::set_symmetric_difference(a1, a1 + SIZE1,
59 a3, a3 + SIZE2, symmetric_difference);
60 cout << "\n\nset_symmetric_difference of a1 and a3 is: ";
61 std::copy(symmetric_difference, ptr, output);
62
63 int unionSet[SIZE3];
64
65 // determina elementos que estão em qualquer conjunto ou em ambos
66 ptr = std::set_union(a1, a1 + SIZE1, a3, a3 + SIZE2, unionSet);
67 cout << "\n\nset_union of a1 and a3 is: ";
68 std::copy(unionSet, ptr, output);
69 cout << endl;
70 return 0;
71 } // fim de main

```

```

a1 contains: 1 2 3 4 5 6 7 8 9 10
a2 contains: 4 5 6 7 8
a3 contains: 4 5 6 11 15

```

```

a1 includes a2
a1 does not include a3

```

```

set_difference of a1 and a2 is: 1 2 3 9 10

```

```

set_intersection of a1 and a2 is: 4 5 6 7 8

```

```

set_symmetric_difference of a1 and a3 is: 1 2 3 7 8 9 10 11 15

```

```

set_union of a1 and a3 is: 1 2 3 4 5 6 7 8 9 10 11 15

```

Figura 23.35 Operações set da Standard Library.

(continuação)

ordem em que os elementos foram originalmente classificados. As duas seqüências devem ser classificadas utilizando a mesma função de comparação.

A linha 66 utiliza a função `set_union` para criar um conjunto de todos os elementos que estão em qualquer um dos dois conjuntos classificados ou em ambos (ambos os conjuntos de valores devem estar na ordem crescente). Os elementos são copiados de ambos os conjuntos para o quinto argumento (nesse caso, o array `unionSet`). Os elementos que aparecem em ambos os conjuntos só são copiados do primeiro conjunto. Os dois primeiros argumentos de iterador devem ser pelo menos iteradores de entrada para o primeiro conjunto de valores. Os dois próximos argumentos de iterador devem ser pelo menos iteradores de entrada para o segundo conjunto de valores. O quinto argumento deve ser pelo menos um iterador de saída que indica onde armazenar os elementos copiados. A função retorna um iterador de saída posicionado imediatamente depois do último valor copiado no conjunto para o qual o quinto argumento aponta. Uma segunda versão de função `set_union` aceita um sexto argumento que é uma função predicado binária para indicar a ordem em que os elementos foram originalmente classificados. As duas seqüências devem ser classificadas utilizando a mesma função de comparação.

### 23.5.11 `lower_bound`, `upper_bound` e `equal_range`

A Figura 23.36 demonstra as funções `lower_bound`, `upper_bound` e `equal_range` da Standard Library. A linha 24 utiliza a função `lower_bound` para encontrar a primeira localização em uma seqüência classificada de valores em que o terceiro argumento poderia ser inserido na seqüência de tal modo que a seqüência ainda seria classificada na ordem crescente. Os dois primeiros argumentos de iterador devem ser pelo menos iteradores de leitura direta. O terceiro argumento é o valor para o qual determinar o limite inferior. A função retorna um iterador de leitura direta que aponta para a posição em que a inserção pode ocorrer. Uma segunda versão da função `lower_bound` aceita como quarto argumento uma função predicado binária indicando a ordem em que os elementos foram originalmente classificados.

A linha 30 usa a função `upper_bound` para encontrar a última localização em uma seqüência classificada de valores em que o terceiro argumento poderia ser inserido na seqüência de tal modo que a seqüência ainda seria classificada na ordem crescente. Os dois primeiros argumentos de iterador devem ser pelo menos iteradores de leitura direta. O terceiro argumento é o valor para o qual determinar o limite

superior. A função retorna um iterador de leitura direta que aponta para a posição em que a inserção pode ocorrer. Uma segunda versão da função `upper_bound` aceita como um quarto argumento uma função predicado binária indicando a ordem em que os elementos foram originalmente classificados.

A linha 38 utiliza a função `equal_range` para retornar um par de iteradores de leitura direta contendo os resultados combinados de realizar tanto a operação `lower_bound` como a `upper_bound`. Os dois primeiros argumentos de iterador devem ser pelo menos iteradores de leitura direta. O terceiro argumento é o valor para o qual localizar o intervalo igual. A função retorna um par de iteradores de leitura direta para o limite inferior (`eq.first`) e para o limite superior (`eq.second`), respectivamente.

As funções `lower_bound`, `upper_bound` e `equal_range` costumam ser utilizadas para localizar pontos de inserção em seqüências classificadas. A linha 47 utiliza `lower_bound` para localizar o primeiro ponto em que 5 pode ser inserido na ordem em `v`. A linha 54 utiliza `upper_bound` para localizar o último ponto em que 7 pode ser inserido na ordem em `v`. A linha 62 utiliza `equal_range` para localizar o primeiro e o último ponto em que 5 pode ser inserido na ordem em `v`.

```

1 // Figura 23.36: Fig23_36.cpp
2 // Funções lower_bound, upper_bound e
3 // equal_range da Standard Library para uma seqüência classificada de valores.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include <algorithm> // definições de algoritmo
9 #include <vector> // definição do template de classe vector
10 #include <iterator> // ostream_iterator
11
12 int main()
13 {
14 const int SIZE = 10;
15 int a1[SIZE] = { 2, 2, 4, 4, 4, 6, 6, 6, 6, 8 };
16 std::vector< int > v(a1, a1 + SIZE); // cópia de a1
17 std::ostream_iterator< int > output(cout, " ");
18
19 cout << "Vector v contains:\n";
20 std::copy(v.begin(), v.end(), output);
21
22 // determina o ponto de inserção no limite inferior para 6 em v
23 std::vector< int >::iterator lower;
24 lower = std::lower_bound(v.begin(), v.end(), 6);
25 cout << "\n\nLower bound of 6 is element "
26 << (lower - v.begin()) << " of vector v";
27
28 // determina o ponto de inserção no limite superior para 6 em v
29 std::vector< int >::iterator upper;
30 upper = std::upper_bound(v.begin(), v.end(), 6);
31 cout << "\n\nUpper bound of 6 is element "
32 << (upper - v.begin()) << " of vector v";
33
34 // utiliza equal_range para determinar os pontos de inserção tanto para o
35 // limite inferior como para o limite superior para 6
36 std::pair< std::vector< int >::iterator,
37 std::vector< int >::iterator > eq;
38 eq = std::equal_range(v.begin(), v.end(), 6);
39 cout << "\n\nUsing equal_range:\n Lower bound of 6 is element "
40 << (eq.first - v.begin()) << " of vector v";
41 cout << "\n Upper bound of 6 is element "
42 << (eq.second - v.begin()) << " of vector v";
43 cout << "\n\nUse lower_bound to locate the first point\n"

```

**Figura 23.36** Algoritmos `lower_bound`, `upper_bound` e `equal_range`.

(continua)

```

44 << "at which 5 can be inserted in order";
45
46 // determina o ponto de inserção no limite inferior para 5 em v
47 lower = std::lower_bound(v.begin(), v.end(), 5);
48 cout << "\n Lower bound of 5 is element "
49 << (lower - v.begin()) << " of vector v";
50 cout << "\n\nUse upper_bound to locate the last point\n"
51 << "at which 7 can be inserted in order";
52
53 // determina o ponto de inserção no limite superior para 7 em v
54 upper = std::upper_bound(v.begin(), v.end(), 7);
55 cout << "\n Upper bound of 7 is element "
56 << (upper - v.begin()) << " of vector v";
57 cout << "\n\nUse equal_range to locate the first and\n"
58 << "last point at which 5 can be inserted in order";
59
60 // utiliza equal_range para determinar os pontos de inserção tanto para o
61 // limite inferior como para o limite superior para 5
62 eq = std::equal_range(v.begin(), v.end(), 5);
63 cout << "\n Lower bound of 5 is element "
64 << (eq.first - v.begin()) << " of vector v";
65 cout << "\n Upper bound of 5 is element "
66 << (eq.second - v.begin()) << " of vector v" << endl;
67 return 0;
68 } // fim do main

```

Vector v contains:

2 2 4 4 4 6 6 6 8

Lower bound of 6 is element 5 of vector v

Upper bound of 6 is element 9 of vector v

Using equal\_range:

Lower bound of 6 is element 5 of vector v

Upper bound of 6 is element 9 of vector v

Use lower\_bound to locate the first point

at which 5 can be inserted in order

Lower bound of 5 is element 5 of vector v

Use upper\_bound to locate the last point

at which 7 can be inserted in order

Upper bound of 7 is element 9 of vector v

Use equal\_range to locate the first and

last point at which 5 can be inserted in order

Lower bound of 5 is element 5 of vector v

Upper bound of 5 is element 5 of vector v

**Figura 23.36** Algoritmos lower\_bound, upper\_bound e equal\_range.

(continuação)

### 23.5.12 Heapsort

A Figura 23.37 demonstra as funções da Standard Library para realizar o **algoritmo de classificação heapsort**. **Heapsort** é um algoritmo de classificação em que um array de elementos é organizado em uma árvore binária especial chamada **heap**. As principais características de um heap são que o maior elemento está sempre na parte superior do heap e os valores dos filhos de qualquer nó na árvore binária são sempre menores que ou iguais ao valor desse nó. Um heap organizado dessa maneira costuma ser chamado de **maxheap**. O heapsort é discutido em detalhes em cursos de ciência da computação chamados ‘Estrutura de dados’ e ‘Algoritmos’.

A linha 23 utiliza a função `make_heap` para aceitar uma sequência de valores no intervalo de `v.begin()` até, mas não incluindo, `v.end()` e cria um heap que pode ser utilizado para produzir uma sequência classificada. Os dois argumentos de iterador devem ser iteradores de acesso aleatório, então essa função funcionará apenas com arrays, vectors e deque. Uma segunda versão dessa função aceita como um terceiro argumento uma função predicado binária para comparar valores.

A linha 27 utiliza a função `sort_heap` para classificar uma sequência de valores no intervalo de `v.begin()` até, mas não incluindo, `v.end()` que já estão organizados em um heap. Os dois argumentos de iterador devem ser iteradores de acesso aleatório. Uma segunda versão dessa função aceita como um terceiro argumento uma função predicado binária para comparar valores.

A linha 41 utiliza a função `push_heap` para adicionar um novo valor em um heap. Pegamos um elemento do array a por vez, acrescentamos esse elemento ao fim de vector `v2` e realizamos a operação `push_heap`. Se o elemento acrescentado for o único elemento no vector, este já é um heap. Caso contrário, a função `push_heap` reorganiza os elementos do vector em um heap. Toda vez que `push_heap` é chamado, ele pressupõe que o último elemento atualmente no vector (isto é, o que é acrescentado antes da chamada de

```

1 // Figura 23.37: Fig23_37.cpp
2 // Algoritmos da Standard Library push_heap, pop_heap,
3 // make_heap e sort_heap.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 #include <algorithm>
9 #include <vector>
10 #include <iterator>
11
12 int main()
13 {
14 const int SIZE = 10;
15 int a[SIZE] = { 3, 100, 52, 77, 22, 31, 1, 98, 13, 40 };
16 std::vector< int > v(a, a + SIZE); // cópia de a
17 std::vector< int > v2;
18 std::ostream_iterator< int > output(cout, " ");
19
20 cout << "Vector v before make_heap:\n";
21 std::copy(v.begin(), v.end(), output);
22
23 std::make_heap(v.begin(), v.end()); // cria o heap a partir do vetor v
24 cout << "\nVector v after make_heap:\n";
25 std::copy(v.begin(), v.end(), output);
26
27 std::sort_heap(v.begin(), v.end()); // classifica elementos com sort_heap
28 cout << "\nVector v after sort_heap:\n";
29 std::copy(v.begin(), v.end(), output);
30
31 // realiza o heapsort com push_heap e pop_heap
32 cout << "\n\nArray a contains: ";
33 std::copy(a, a + SIZE, output); // exibe array a
34 cout << endl;
35
36 // coloca elementos de array a em v2 e
37 // coloca elementos de v2 em heap
38 for (int i = 0; i < SIZE; i++)
39 {
40 v2.push_back(a[i]);
41 std::push_heap(v2.begin(), v2.end());
42 cout << "\nv2 after push_heap(a[" << i << "]): ";
43 std::copy(v2.begin(), v2.end(), output);

```

**Figura 23.37** Utilizando funções Standard Library para realizar um heapsort.

(continua)



```

44 } // fim do for
45
46 cout << endl;
47
48 // remove elementos de heap na ordem classificada
49 for (unsigned int j = 0; j < v2.size(); j++)
50 {
51 cout << "\nv2 after " << v2[0] << " popped from heap\n";
52 std::pop_heap(v2.begin(), v2.end() - j);
53 std::copy(v2.begin(), v2.end(), output);
54 } // fim do for
55
56 cout << endl;
57 return 0;
58 } // fim de main

```

Vector v before make\_heap:

3 100 52 77 22 31 1 98 13 40

Vector v after make\_heap:

100 98 52 77 40 31 1 3 13 22

Vector v after sort\_heap:

1 3 13 22 31 40 52 77 98 100

Array a contains: 3 100 52 77 22 31 1 98 13 40

v2 after push\_heap(a[0]): 3

v2 after push\_heap(a[1]): 100 3

v2 after push\_heap(a[2]): 100 3 52

v2 after push\_heap(a[3]): 100 77 52 3

v2 after push\_heap(a[4]): 100 77 52 3 22

v2 after push\_heap(a[5]): 100 77 52 3 22 31

v2 after push\_heap(a[6]): 100 77 52 3 22 31 1

v2 after push\_heap(a[7]): 100 98 52 77 22 31 1 3

v2 after push\_heap(a[8]): 100 98 52 77 22 31 1 3 13

v2 after push\_heap(a[9]): 100 98 52 77 40 31 1 3 13 22

v2 after 100 popped from heap

98 77 52 22 40 31 1 3 13 100

v2 after 98 popped from heap

77 40 52 22 13 31 1 3 98 100

v2 after 77 popped from heap

52 40 31 22 13 3 1 77 98 100

v2 after 52 popped from heap

40 22 31 1 13 3 52 77 98 100

v2 after 40 popped from heap

31 22 3 1 13 40 52 77 98 100

v2 after 31 popped from heap

22 13 3 1 31 40 52 77 98 100

v2 after 22 popped from heap

13 1 3 22 31 40 52 77 98 100

v2 after 13 popped from heap

3 1 13 22 31 40 52 77 98 100

v2 after 3 popped from heap

1 3 13 22 31 40 52 77 98 100

v2 after 1 popped from heap

1 3 13 22 31 40 52 77 98 100

**Figura 23.37** Utilizando funções Standard Library para realizar um heapsort.

(continuação)

função `push_heap`) é o elemento sendo adicionado ao heap e que todos os outros elementos no vector já estão organizados como um heap. Os dois argumentos de iterador para `push_heap` devem ser iteradores de acesso aleatório. Uma segunda versão dessa função aceita como um terceiro argumento uma função predicado binária para comparar valores.

A linha 52 utiliza `pop_heap` para remover o elemento superior do heap. Essa função pressupõe que os elementos no intervalo especificado por seus dois argumentos de acesso aleatório de iterador já são um heap. Remover repetidamente o elemento superior do heap resulta em uma sequência classificada de valores. A função `pop_heap` permuta o primeiro elemento do heap (`v2.begin()`, nesse exemplo) pelo último elemento do heap (o elemento antes de `v2.end() - i`, nesse exemplo), então assegura que os elementos até, mas não incluindo, o último elemento, ainda formam um heap. Note na saída que, depois das operações `pop_heap`, o vector é classificado em ordem crescente. Uma segunda versão dessa função aceita como um terceiro argumento uma função predicado binária para comparar valores.

### 23.5.13 min e max

Os algoritmos `min` e `max` determinam o mínimo e o máximo de dois elementos, respectivamente. A Figura 23.38 demonstra `min` e `max` para valores `int` e `char`.

### 23.5.14 Algoritmos de STL não discutidos neste capítulo

A Figura 23.39 resume os algoritmos STL que não são abordados neste capítulo.

```

1 // Figura 23.38: Fig23_38.cpp
2 // Algoritmos min e max da Standard Library.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <algorithm>
8
9 int main()
10 {
11 cout << "The minimum of 12 and 7 is: " << std::min(12, 7);
12 cout << "\nThe maximum of 12 and 7 is: " << std::max(12, 7);
13 cout << "\nThe minimum of 'G' and 'Z' is: " << std::min('G', 'Z');
14 cout << "\nThe maximum of 'G' and 'Z' is: " << std::max('G', 'Z');
15 cout << endl;
16 return 0;
17 } // fim de main

```

```

The minimum of 12 and 7 is: 7
The maximum of 12 and 7 is: 12
The minimum of 'G' and 'Z' is: G
The maximum of 'G' and 'Z' is: Z

```

Figura 23.38 Algoritmos `min` e `max`.

Algoritmo	Descrição
<code>inner_product</code>	Calcula a soma dos produtos de duas seqüências aceitando os elementos correspondentes em cada seqüência, multiplicando esses elementos e adicionando o resultado a um total.
<code>adjacent_difference</code>	Começando com o segundo elemento em uma seqüência, calcula a diferença (usando o operador <code>-</code> ) entre os elementos atuais e anteriores e armazena o resultado. Os dois primeiros argumentos de iterador de entrada indicam o intervalo de elementos no contêiner e o terceiro indica onde os resultados devem ser armazenados. Uma segunda versão desse algoritmo aceita como quarto argumento uma função binária para realizar um cálculo entre o elemento atual e o elemento anterior.

Figura 23.39 Algoritmos não abordados neste capítulo.

(continua)

Algoritmo	Descrição
<code>partial_sum</code>	Calcula a soma total (usando o operador <code>+</code> ) dos valores em uma sequência. Os dois primeiros argumentos de iterador de entrada indicam o intervalo de elementos no contêiner e o terceiro indica onde os resultados devem ser armazenados. Uma segunda versão desse algoritmo aceita como quarto argumento uma função binária que realiza um cálculo entre o valor atual na sequência e a soma total.
<code>nth_element</code>	Utiliza três iteradores de acesso aleatório para particionar um intervalo de elementos. O primeiro e o último argumentos representam o intervalo de elementos. O segundo argumento é a localização do elemento divisor. Depois que esse algoritmo executar, todos os elementos antes do elemento divisor são menores que aquele elemento e todos os elementos depois do elemento divisor são maiores que ou igual àquele elemento. Uma segunda versão desse algoritmo aceita como quarto argumento uma função binária de comparação.
<code>partition</code>	Este algoritmo é semelhante a <code>nth_element</code> , mas requer iteradores bidirecionais menos poderosos, tornando-o mais flexível que <code>nth_element</code> . O algoritmo <code>partition</code> requer dois iteradores bidirecionais que indicam o intervalo de elementos para partição. O terceiro elemento é uma função predicado unária que ajuda a particionar os elementos para que todos os elementos na sequência pela qual o predicado é <code>true</code> estejam à esquerda (em direção ao início da sequência) de todos os elementos para os quais o predicado é <code>false</code> . Um iterador bidirecional é retornado indicando o primeiro elemento na sequência pela qual o predicado retorna <code>false</code> .
<code>stable_partition</code>	Este algoritmo é semelhante a <code>partition</code> , exceto pelo fato de que os elementos pelos quais a função predicado retorna <code>true</code> são mantidos em sua ordem original e os elementos pelos quais a função predicado retorna <code>false</code> são mantidos em sua ordem original.
<code>next_permutation</code>	Permutação lexicográfica seguinte de uma sequência.
<code>prev_permutation</code>	Permutação lexicográfica anterior de uma sequência.
<code>rotate</code>	Utiliza três argumentos de iterador de leitura direta para rotacionar a sequência indicada pelo primeiro e pelo último argumento pelo número de posições indicado subtraindo o primeiro argumento do segundo argumento. Por exemplo, a sequência 1, 2, 3, 4, 5 rotacionada por duas posições seria 4, 5, 1, 2, 3.
<code>rotate_copy</code>	Este algoritmo é idêntico a <code>rotate</code> , exceto que os resultados são armazenados em uma sequência separada indicada pelo quarto argumento — um iterador de saída. As duas sequências devem ter o mesmo número de elementos.
<code>adjacent_find</code>	Este algoritmo retorna um iterador de entrada que indica o primeiro de dois elementos adjacentes idênticos em uma sequência. Se não houver elementos adjacentes idênticos, o iterador é posicionado no <code>end</code> da sequência.
<code>search</code>	Este algoritmo procura uma subsequência de elementos dentro de uma sequência de elementos <code>e</code> , se tal subsequência é localizada, retorna um iterador de leitura direta que indica o primeiro elemento dessa subsequência. Se não houver nenhuma correspondência, o iterador é posicionado no fim da sequência a ser pesquisada.
<code>search_n</code>	Este algoritmo pesquisa uma sequência de elementos procurando uma subsequência em que os valores de um número especificado de elementos têm um valor particular <code>e</code> , se tal subsequência é localizada, retorna um iterador de leitura direta que indica o primeiro elemento dessa subsequência. Se não houver nenhuma correspondência, o iterador é posicionado no fim da sequência a ser pesquisada.
<code>partial_sort</code>	Utiliza três iteradores de acesso aleatório para classificar parte de uma sequência. O primeiro e o último argumento indicam a sequência de elementos. O segundo argumento indica a localização final para a parte classificada da sequência. Por padrão, os elementos são ordenados usando o operador <code>&lt;</code> (uma função predicado binária também pode ser fornecida). Os elementos do segundo iterador de argumento para o fim da sequência estão em uma ordem indefinida.
<code>partial_sort_copy</code>	Utiliza dois iteradores de entrada e dois iteradores de acesso aleatório para classificar parte da sequência indicada pelos dois argumentos de iterador de entrada. Os resultados são armazenados na sequência indicada pelos dois argumentos de iterador de acesso aleatório. Por padrão, os elementos são ordenados usando o operador <code>&lt;</code> (uma função predicado binária também pode ser fornecida). O número de elementos classificado é o menor do número de elementos no resultado e o número de elementos na sequência original.
<code>stable_sort</code>	O algoritmo é semelhante a <code>sort</code> , exceto pelo fato de que todos os elementos iguais são mantidos em sua ordem original.

**Figura 23.39** Algoritmos não abordados neste capítulo.

(continuação)

## 23.6 Classe `bitset`

A classe `bitset` facilita a criação e manipulação de **conjuntos de bits**, que são úteis para representar um conjunto de flags de bit. `bitsets` têm tamanho definido em tempo de compilação. A classe `bitset` é uma ferramenta alternativa para manipulação de bits, discutida no Capítulo 22. A declaração

```
bitset< size > b;
```

cria o `bitset` `b`, em que cada bit é inicialmente 0. A instrução

```
b.set(bitNumber);
```

configura o bit `bitNumber` do `bitset` `b` como 'on' (ativado). A expressão `b.set()` configura todos os bits em `b` como 'on'.

A instrução

```
b.reset(bitNumber);
```

configura o bit `bitNumber` do `bitset` `b` como 'off' (desativado). A expressão `b.reset()` configura todos os bits em `b` como 'off' (desativado). A instrução

```
b.flip(bitNumber);
```

inverte o bit `bitNumber` do `bitset` `b` (por exemplo, se o bit estiver ativado, `flip` o desativa). A expressão `b.flip()` inverte todos os bits em `b`. A instrução

```
b[bitNumber];
```

retorna uma referência ao bit `bitNumber` de `bitset` `b`. De maneira semelhante,

```
b.at(bitNumber);
```

realiza a verificação de intervalos em `bitNumber` primeiro. Então, se `bitNumber` estiver no intervalo, `at` retorna uma referência ao bit. Caso contrário, `at` lança uma exceção `out_of_range`. A instrução

```
b.test(bitNumber);
```

realiza a verificação de intervalos em `bitNumber` primeiro. Então, se `bitNumber` estiver no intervalo, `test` retorna `true` se o bit estiver ativado, `false` se o bit estiver desativado. Caso contrário, `test` lança uma exceção `out_of_range`. A expressão

```
b.size()
```

retorna o número de bits em `bitset` `b`. A expressão

```
b.count()
```

retorna o número de bits que estão configurados em `bitset` `b`. A expressão

```
b.any()
```

retorna `true` se qualquer bit estiver configurado em `bitset` `b`. A expressão

```
b.none()
```

retorna `true` se nenhum dos bits estiver configurado em `bitset` `b`. As expressões

```
b == b1
```

```
b != b1
```

comparam a igualdade e desigualdade dos dois `bitsets`, respectivamente.

Cada um dos operadores de atribuição de bit `&=`, `|=` e `^=` pode ser utilizado para combinar `bitsets`. Por exemplo,

```
b &= b1;
```

realiza uma operação E lógica de bit a bit entre os `bitsets` `b` e `b1`. O resultado é armazenado em `b`. O OR lógico de bitwise e o XOR lógico de bitwise são realizados por

```
b |= b1;
```

```
b ^= b2;
```

A expressão

```
b >>= n;
```

desloca os bits em `bitset` `b` para a direita por `n` posições. A expressão

```
b <<= n;
```

desloca os bits em `bitset` `b` para a esquerda por `n` posições. As expressões

```
b.to_string()
```

```
b.to_ulong()
```

convertem `bitset` `b` em uma `string` e em um `unsigned long`, respectivamente.

### Crivo de Eratóstenes com `bitset`

A Figura 23.40 examina novamente o Crivo de Eratóstenes para localizar números primos que discutimos no Exercício 7.29. Um `bitset` é utilizado em vez de um array para implementar o algoritmo. O programa exhibe todos os números primos de 2 a 1.023, e em seguida permite que o usuário insira um número para determinar se esse número é primo.

A linha 20 cria um `bitset` de `size` bits (`size` é 1.024 nesse exemplo). Por padrão, todos os bits no `bitset` são configurados como ‘off’ (desativados). A linha 21 chama a função `flip` para configurar todos os bits como ‘on’ (ativados). Os números 0 e 1 não são números primos, então as linhas 22–23 chamam a função `reset` para configurar os bits 0 e 1 como ‘off’. As linhas 29–36 determinam todos os números primos de 2 a 1.023. O inteiro `finalBit` (linha 26) é utilizado para determinar quando o algoritmo está completo. O algoritmo básico diz que um número é primo se ele não tiver nenhum divisor diferente de 1 e dele próprio. Iniciando com o número 2, podemos eliminar todos os múltiplos desse número. O número 2 é divisível somente por 1 e por ele próprio, então é primo. Portanto, podemos eliminar 4, 6, 8 e assim por diante. O número 3 é divisível somente por 1 e por si próprio. Portanto, podemos eliminar todos os múltiplos de 3 (tenha em mente que todos os números pares já foram eliminados).

```

1 // Figura 23.40: Fig23_40.cpp
2 // Utilizando um bitset para demonstrar o Crivo de Eratóstenes.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6 using std::endl;
7
8 #include <iomanip>
9 using std::setw;
10
11 #include <cmath>
12 using std::sqrt; // protótipo de sqrt
13
14 #include <bitset> // definição da classe bitset
15
16 int main()
17 {
18 const int SIZE = 1024;
19 int value;
20 std::bitset< SIZE > sieve; // cria bitset de 1024 bits
21 sieve.flip(); // inverte todos os bits no crivo de bitset
22 sieve.reset(0); // redefine o primeiro bit (número 0)
23 sieve.reset(1); // redefine o segundo bit (número 1)
24
25 // realiza o Crivo de Eratóstenes
26 int finalBit = sqrt(static_cast< double > (sieve.size())) + 1;
27
28 // determina todos os números primos de 2 a 1024
29 for (int i = 2; i < finalBit; i++)
30 {
31 if (sieve.test(i)) // bit i está ativado
32 {
33 for (int j = 2 * i; j < SIZE; j += i)
34 sieve.reset(j); // configurado bit j como desativado
35 } // fim do if
36 } // fim do for
37
38 cout << "The prime numbers in the range 2 to 1023 are:\n";
39
40 // exhibe números primos no intervalo 2-1023
41 for (int k = 2, counter = 1; k < SIZE; k++)

```

**Figura 23.40** A classe `bitset` e o crivo de Eratóstenes.

(continua)

```

42 {
43 if (sieve.test(k)) // o bit k está ativado
44 {
45 cout << setw(5) << k;
46
47 if (counter++ % 12 == 0) // contador é um múltiplo de 12
48 cout << '\n';
49 } // fim do if
50 } // fim do for
51
52 cout << endl;
53
54 // obtém o valor inserido pelo usuário para determinar se valor é ou não primo
55 cout << "\nEnter a value from 2 to 1023 (-1 to end): ";
56 cin >> value;
57
58 // determina se a entrada de usuário é ou não um número primo
59 while (value != -1)
60 {
61 if (sieve[value]) // número primo
62 cout << value << " is a prime number\n";
63 else // não um número primo
64 cout << value << " is not a prime number\n";
65
66 cout << "\nEnter a value from 2 to 1023 (-1 to end): ";
67 cin >> value;
68 } // fim do while
69
70 return 0;
71 } // fim de main

```

The prime numbers in the range 2 to 1023 are:

```

 2 3 5 7 11 13 17 19 23 29 31 37
41 43 47 53 59 61 67 71 73 79 83 89
97 101 103 107 109 113 127 131 137 139 149 151
157 163 167 173 179 181 191 193 197 199 211 223
227 229 233 239 241 251 257 263 269 271 277 281
283 293 307 311 313 317 331 337 347 349 353 359
367 373 379 383 389 397 401 409 419 421 431 433
439 443 449 457 461 463 467 479 487 491 499 503
509 521 523 541 547 557 563 569 571 577 587 593
599 601 607 613 617 619 631 641 643 647 653 659
661 673 677 683 691 701 709 719 727 733 739 743
751 757 761 769 773 787 797 809 811 821 823 827
829 839 853 857 859 863 877 881 883 887 907 911
919 929 937 941 947 953 967 971 977 983 991 997
1009 1013 1019 1021

```

Enter a value from 2 to 1023 (-1 to end): **389**  
389 is a prime number

Enter a value from 2 to 1023 (-1 to end): **88**  
88 is not a prime number

Enter a value from 2 to 1023 (-1 to end): **-1**

**Figura 23.40** A classe bitset e o crivo de Eratóstenes.

(continuação)

## 23.7    Objetos de função

Muitos algoritmos STL permitem passar um ponteiro de função no algoritmo para ajudar o algoritmo a realizar sua tarefa. Por exemplo, o algoritmo `binary_search` que discutimos na Seção 23.5.6 é sobrecarregado com uma versão que requer como seu quarto parâmetro um ponteiro para uma função que aceita dois argumentos e retorna um valor `bool`. O algoritmo `binary_search` utiliza essa função para comparar a chave de pesquisa com um elemento na coleção. A função retorna `true` se a chave de pesquisa e o elemento sob comparação forem iguais; caso contrário, a função retorna `false`. Isso permite que `binary_search` pesquise uma coleção de elementos para os quais o tipo de elemento não fornece um operador `==` de igualdade sobrecarregado.

Os designers STL tornaram os algoritmos mais flexíveis permitindo a qualquer algoritmo que pode receber um ponteiro de função receber um objeto de uma classe que sobrecarrega o operador de parênteses com uma função chamada `operator()`, contanto que o operador sobrecarregado atenda aos requisitos do algoritmo — no caso de `binary_search`, ele deve receber dois argumentos e retornar um `bool`. Um objeto de tal classe é conhecido como um **objeto de função** e pode ser utilizado sintática e semanticamente como uma função ou ponteiro de função — o operador parênteses sobrecarregado é invocado utilizando o nome de um objeto de função seguido por parênteses contendo os argumentos para a função.

Os objetos de função fornecem várias vantagens sobre os ponteiros de função. Visto que os objetos de função são comumente implementados como templates de classe que são incluídos em cada arquivo de código-fonte que as utiliza, o compilador pode colocar inline um `operator()` sobrecarregado para melhorar o desempenho. Além disso, visto que são objetos de classes, os objetos de função podem ter membros de dados que o `operator()` pode utilizar para realizar sua tarefa.

### *Objetos de função predefinidos da Standard Template Library*

Muitos objetos de função predefinidos podem ser encontrados no cabeçalho `<functional>`. A Figura 23.41 lista vários dos objetos de função STL, que são todos implementados como templates de classe. Utilizamos o objeto de função `less< T >` nos exemplos `set`, `multiset` e `priority_queue`, para especificar a ordem de classificação para elementos em um contêiner.

### *Utilizando o algoritmo **Accumulate** STL*

A Figura 23.42 demonstra o algoritmo numérico `accumulate` (discutido na Figura 23.30) para calcular a soma dos quadrados dos elementos em um vetor. O quarto argumento para `accumulate` é um **objeto de função binária** (isto é, um objeto de função para o qual `operator()` aceita dois argumentos) ou um ponteiro de função para uma **função binária** (isto é, uma função que aceita dois argumentos). A função `accumulate` é demonstrada duas vezes — uma vez com um ponteiro de função e, outra, com um objeto de função.

As linhas 15–18 definem uma função `sumSquares` que eleva ao quadrado seu segundo argumento `value`, soma esse quadrado e seu primeiro argumento `total` e retorna o resultado. A função `accumulate` passará cada um dos elementos da sequência pela qual ela itera como o segundo argumento para `sumSquares` no exemplo. Na primeira chamada para `sumSquares`, o primeiro argumento será o valor inicial do `total` (que é fornecido como o terceiro argumento para `accumulate`; 0 nesse programa). Todas as chamadas subsequentes a `sumSquares` recebem como o primeiro argumento a soma parcial retornada pela chamada anterior a `sumSquares`. Quando `accumulate` completa, ela retorna a soma dos quadrados de todos os elementos na sequência.

As linhas 23–32 definem uma classe `SumSquaresClass` que herda do template da classe `binary_function` (no arquivo de cabeçalho `<functional>`) — uma classe básica vazia para criar objetos de função em que `operator` recebe dois parâmetros e retorna um valor. A classe `binary_function` aceita três parâmetros de tipo que representam os tipos do primeiro argumento, do segundo argumento e do valor de retorno de `operator`, respectivamente. Nesse exemplo, o tipo desses parâmetros é `T` (linha 24). Na primeira chamada para o objeto de função, o primeiro argumento será o valor inicial do `total` (que é fornecido como o terceiro argumento para `accumulate`; 0

Objetos de função STL	Tipo	Objetos de função STL	Tipo
<code>divides&lt; T &gt;</code>	aritmético	<code>logical_or&lt; T &gt;</code>	lógico
<code>equal_to&lt; T &gt;</code>	relacional	<code>minus&lt; T &gt;</code>	aritmético
<code>greater&lt; T &gt;</code>	relacional	<code>modulus&lt; T &gt;</code>	aritmético
<code>greater_equal&lt; T &gt;</code>	relacional	<code>negate&lt; T &gt;</code>	aritmético
<code>less&lt; T &gt;</code>	relacional	<code>not_equal_to&lt; T &gt;</code>	relacional
<code>less_equal&lt; T &gt;</code>	relacional	<code>plus&lt; T &gt;</code>	aritmético
<code>logical_and&lt; T &gt;</code>	lógico	<code>multiplies&lt; T &gt;</code>	aritmético
<code>logical_not&lt; T &gt;</code>	lógico		

**Figura 23.41**    Objetos de função na Standard Library.



```

1 // Figura 23.42: Fig23_42.cpp
2 // Demonstrando objetos de função.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <vector> // definição do template de classe vector
8 #include <algorithm> // algoritmo copy
9 #include <numeric> // algoritmo accumulate
10 #include <functional> // definição binary_function
11 #include <iterator> // ostream_iterator
12
13 // função binária soma o quadrado de seu segundo argumento e o
14 // total parcial em seu primeiro argumento, e retorna a soma
15 int sumSquares(int total, int value)
16 {
17 return total + value * value;
18 } // fim da função sumSquares
19
20 // template da classe de função binária define o operator() sobrecarregado
21 // isso adiciona o quadrado de seu segundo argumento e o total
22 // em seu primeiro argumento, então retorna a soma
23 template< typename T >
24 class SumSquaresClass : public std::binary_function< T, T, T >
25 {
26 public:
27 // adiciona o quadrado de value a total e retorna o resultado
28 T operator()(const T &total, const T &value)
29 {
30 return total + value * value;
31 } // fim da função operator()
32 }; // fim da classe SumSquaresClass
33
34 int main()
35 {
36 const int SIZE = 10;
37 int array[SIZE] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
38 std::vector< int > integers(array, array + SIZE); // cópia de array
39 std::ostream_iterator< int > output(cout, " ");
40 int result;
41
42 cout << "vector integers contains:\n";
43 std::copy(integers.begin(), integers.end(), output);
44
45 // calcula soma de quadrados dos elementos do vector integers
46 // utilizando a função binária sumSquares
47 result = std::accumulate(integers.begin(), integers.end(),
48 0, sumSquares);
49
50 cout << "\n\nSum of squares of elements in integers using "
51 << "binary\function sumSquares: " << result;
52
53 // calcula soma de quadrados de elementos de inteiros de vetor
54 // utilizando o objeto de função binária
55 result = std::accumulate(integers.begin(), integers.end(),
56 0, SumSquaresClass< int >());

```

Figura 23.42 Objeto de função binária.

(continua)

```

57
58 cout << "\n\nSum of squares of elements in integers using "
59 << "binary\nfunction object of type "
60 << "SumSquaresClass< int >: " << result << endl;
61 return 0;
62 } // fim do main

```

```

vector integers contains:
1 2 3 4 5 6 7 8 9 10

```

```

Sum of squares of elements in integers using binary
function sumSquares: 385

```

```

Sum of squares of elements in integers using binary
function object of type SumSquaresClass< int >: 385

```

**Figura 23.42** Objeto de função binária.

(continuação)

nesse programa) e o segundo argumento será o primeiro elemento em `vector integers`. Todas as chamadas subsequentes para `operator` recebem como primeiro argumento o resultado retornado pela chamada anterior para o objeto de função, e o segundo argumento será o próximo elemento no `vector`. Quando completar, `accumulate` retornará a soma dos quadrados de todos os elementos no `vector`.

As linhas 47–48 chamam a função `accumulate` com um ponteiro para a função `sumSquares` como seu último argumento.

A instrução nas linhas 55–56 chama a função `accumulate` com um objeto da classe `SumSquaresClass` como o último argumento. A expressão `SumSquaresClass< int >()` cria uma instância da classe `SumSquaresClass` (um objeto de função) que é passado para `accumulate`, que envia a mensagem (invoca a função) de `operator` para o objeto. A instrução poderia ser escrita como duas instruções separadas, assim:

```

SumSquaresClass< int > sumSquaresObject;
result = std::accumulate(integers.begin(), integers.end(),
 0, sumSquaresObject);

```

A primeira linha define um objeto da classe `SumSquaresClass`. Esse objeto é então passado para a função `accumulate`.

## 23.8 Síntese

Neste capítulo, introduzimos a Standard Template Library e discutimos seus três componentes-chave — os contêineres, os iteradores e os algoritmos. Você aprendeu os contêineres de sequência STL, `vector`, `deque` e `list`, que representam as estruturas de dados lineares. Discutimos contêineres associativos, `set`, `multiset`, `map` e `multimap`, que representam as estruturas de dados não-lineares. Você também viu que os adaptadores de contêiner `stack`, `queue` e `priority_queue` podem ser utilizados para limitar as operações dos contêineres de sequência com o propósito de implementar estruturas de dados especializadas representadas pelos adaptadores de contêiner. Então demonstramos muitos dos algoritmos STL, inclusive os algoritmos matemáticos, os algoritmos de pesquisa e classificação básicos e as operações com conjuntos. Você aprendeu os tipos de iteradores que cada algoritmo requer e que cada algoritmo pode ser utilizado com qualquer contêiner que suporte as funcionalidades mínimas de um iterador que o algoritmo requer. Você também aprendeu sobre a classe `bitset`, que facilita a criação e a manipulação de conjuntos de bits como um contêiner. Por fim, introduzimos os objetos de função, que funcionam sintática e semanticamente como funções comuns, mas oferecem vantagens como desempenho e a capacidade de armazenar dados.

No próximo capítulo, discutimos vários dos mais avançados recursos C++, incluindo operadores de coerção, namespaces, palavras-chave de operador, operadores de 'membro de ponteiro para classe', herança múltipla e classes básicas `virtual`.

## 23.9 Recursos sobre C++ na Internet e na Web

A seguir uma coleção de recursos STL na Internet e na World Wide Web. Esses sites incluem tutoriais, referências, FAQs, artigos, livros, entrevistas e software.

### Tutoriais

[www.cs.brown.edu/people/jak/programming/stl-tutorial/tutorial.html](http://www.cs.brown.edu/people/jak/programming/stl-tutorial/tutorial.html)

Este tutorial de STL é organizado por exemplos, filosofia, componentes e estender a STL. Você encontrará exemplos de código que utilizam os componentes STL, explicações úteis e diagramas úteis.

[www.yr1.co.uk/phil/stl/stl.htmlx](http://www.yr1.co.uk/phil/stl/stl.htmlx)

Este tutorial de STL fornece informações sobre templates de função, templates de classe, componentes STL, contêineres, iteradores, adaptadores e objetos de função.

[www.xraylith.wisc.edu/~khan/software/stl/os\\_examples/examples.html](http://www.xraylith.wisc.edu/~khan/software/stl/os_examples/examples.html)

Este site é útil simplesmente para o aprendizado de STL. Você encontrará uma introdução à STL e exemplos do ObjectSpace STL Tool Kit.

### Referências

[www.sgi.com/tech/stl](http://www.sgi.com/tech/stl)

O Silicon Graphics Standard Template Library Programmer's Guide é um recurso útil para obter informações sobre a STL. Você pode fazer download da STL a partir desse site e encontrar as informações mais recentes, documentação de design e links para outros recursos STL.

[www.cppreference.com/cpp\\_stl.html](http://www.cppreference.com/cpp_stl.html)

Este site lista os construtores, operadores e funções suportadas por cada contêiner STL.

### Artigos, livros e entrevistas

[www.byte.com/art/9510/sec12/art3.htm](http://www.byte.com/art/9510/sec12/art3.htm)

O site *Byte Magazine* tem uma cópia de um artigo sobre a STL escrito por Alexander Stepanov. Stepanov, um dos criadores da Standard Template Library, fornece informações sobre o uso da STL na programação genérica.

[www.sgi.com/tech/stl/drdobbs-interview.html](http://www.sgi.com/tech/stl/drdobbs-interview.html)

Uma entrevista do Dr. Dobb's Journal com Alexander Stepanov.

### ANSI/ISO C++ Standard

[www.ansi.org](http://www.ansi.org)

Você pode comprar uma cópia do documento C++ padrão a partir deste site.

### Software

[www.cs.rpi.edu/~musser/stl-book](http://www.cs.rpi.edu/~musser/stl-book)

O site RPI STL inclui informações sobre como a STL difere de outras bibliotecas C++ e sobre como compilar programas que utilizam a STL. O site lista os arquivos STL e fornece programas de exemplo que utilizam STL, classes de contêiner e categorias de iteradores STL. Também fornece uma lista de compiladores compatíveis com STL, sites FTP para código-fonte STL e materiais relacionados.

[msdn.microsoft.com/visualc](http://msdn.microsoft.com/visualc)

Esta é a home page do Microsoft Visual C++. Aqui você pode encontrar as notícias mais recentes sobre o Visual C++, atualizações, recursos técnicos, exemplos e downloads.

[www.borland.com/cbuilder](http://www.borland.com/cbuilder)

Esta é a home page da Borland C++Builder. Aqui você pode encontrar uma variedade de recursos C++, incluindo vários newsgroups de C++, informações sobre os últimos aprimoramentos de produtos, FAQs e muitos outros recursos para programadores que utilizam o C++Builder.

## Resumo

- A Standard Template Library define poderosos componentes reutilizáveis, baseados em template, que implementam muitas estruturas de dados comuns e algoritmos utilizados para processar essas estruturas de dados.
- A STL tem três componentes-chave — contêineres, iteradores e algoritmos.
- Os contêineres STL são estruturas de dados capazes de armazenar objetos de qualquer tipo de dados. Há três categorias de contêiner — contêineres de primeira classe, adaptadores e semicontêineres.
- Os algoritmos STL são funções que realizam manipulações de dados tão comuns como pesquisar, classificar e comparar elementos ou contêineres inteiros.
- Os contêineres são divididos em três categorias importantes — contêineres de sequência, contêineres associativos e adaptadores de contêiner.
- Os contêineres de sequência representam estruturas de dados lineares, como vetores e listas vinculadas.
- Os contêineres associativos são contêineres não-lineares que localizam rapidamente elementos armazenados neles, como conjuntos de valores ou pares chave/valor.
- Os contêineres de sequência e contêineres associativos são coletivamente referidos como contêineres de primeira classe.
- A função de contêiner de primeira classe `begin` retorna um iterador que aponta para o primeiro elemento de um contêiner. A função `end` retorna um iterador que aponta para o primeiro elemento depois do fim do contêiner (um elemento que não existe e, em geral, é utilizado em um loop para indicar quando terminar processamento dos elementos do contêiner).

- Um `istream_iterator` é capaz de extrair valores de uma maneira fortemente tipada a partir de um fluxo de entrada. Um `ostream_iterator` é capaz de inserir valores em um fluxo de saída.
- Iteradores de entrada e de saída podem mover-se somente para a frente (por exemplo, do início para o fim do contêiner) um elemento por vez.
- Um iterador de leitura direta (*forward*) combina as capacidades dos iteradores de entrada e iteradores de saída.
- Um iterador bidirecional tem as capacidades de um iterador de leitura direta e a capacidade de mover-se para trás (isto é, mover-se do fim para o início do contêiner).
- Um iterador de acesso aleatório tem as capacidades de um iterador bidirecional e a capacidade de acessar diretamente qualquer elemento do contêiner.
- Os contêineres que suportam os iteradores de acesso aleatório, como `vector`, podem ser utilizados com todos os algoritmos na STL.
- A STL fornece três contêineres de sequência — `vector`, `list` e `deque`. O template da classe `vector` e o template da classe `deque` são ambos baseados em arrays. O template da classe `list` implementa uma estrutura de dados de lista vinculada.
- A função `capacity` retorna o número de elementos que podem ser armazenados em um vetor antes de o vetor redimensionar-se dinamicamente para acomodar mais elementos.
- A função de contêiner de sequência `push_back` adiciona um elemento ao fim de um contêiner.
- Para utilizar os algoritmos da STL, você deve incluir o arquivo de cabeçalho `<algorithm>`.
- O algoritmo `copy` copia cada elemento em um contêiner que inicia com a localização especificada pelo iterador em seu primeiro argumento e até — mas não incluindo — a localização especificada pelo iterador em seu segundo argumento.
- A função `front` retorna uma referência ao primeiro elemento em um contêiner de sequência. A função `begin` retorna um iterador que aponta para o início de um contêiner de sequência.
- A função `back` retorna uma referência ao último elemento em um contêiner de sequência. A função `end` retorna um iterador que aponta para o elemento um além do fim de um contêiner de sequência.
- A função de contêiner de sequência `insert` insere valor(es) antes do elemento em uma localização específica.
- A função `erase` (disponível em todos os contêineres de primeira classe) remove elemento(s) específico(s) do contêiner.
- A função `empty` (disponível em todos os contêineres e adaptadores) retorna `true` se o contêiner está vazio.
- A função `clear` (disponível em todos os contêineres de primeira classe) esvazia o contêiner.
- O contêiner de sequência `list` fornece uma implementação eficiente para operações de inserção e exclusão em qualquer localização do contêiner. O arquivo de cabeçalho `<list>` deve ser incluído para utilizar o template da classe `list`.
- A função-membro `list::push_front` insere valores no início de uma lista.
- A função-membro `list::sort` organiza os elementos na lista na ordem crescente.
- A função-membro `list::splice` remove elementos em uma `list` e os insere em outra `list` em uma posição específica.
- A função-membro `list::unique` remove elementos duplicados em uma `list`.
- A função-membro `list::assign` substitui o conteúdo de uma `list` pelo conteúdo de outra.
- A função-membro `list::remove` exclui todas as cópias de um valor especificado de uma `list`.
- O template de classe `deque` fornece as mesmas operações que `vector`, mas adiciona as funções-membro `push_front` e `pop_front` para permitir a inserção e exclusão no início de um `deque`, respectivamente. O arquivo de cabeçalho `<deque>` deve ser incluído para utilizar o template da classe `deque`.
- Os contêineres associativos da STL fornecem acesso direto para armazenar e recuperar elementos por meio de chaves.
- Os quatro contêineres associativos são `multiset`, `set`, `multimap` e `map`.
- Os templates de classe `multiset` e `set` fornecem operações para manipular conjuntos de valores onde os valores são as chaves — não há um valor separado associado a cada chave. O arquivo de cabeçalho `<set>` deve ser incluído para utilizar os templates de classe `set` e `multiset`.
- A principal diferença entre um `multiset` e um `set` é que um `multiset` permite chaves duplicadas e um `set` não.
- Os templates de classe `multimap` e `map` fornecem operações para manipular valores associados com chaves.
- A principal diferença entre um `multimap` e um `map` é que um `multimap` permite que chaves duplicadas com valores associados sejam armazenadas e um `map` permite apenas chaves únicas com valores associados.
- A função `count` (disponível em todos os contêineres associativos) conta o número de ocorrências do valor especificado atualmente em um contêiner.
- A função `find` (disponível em todos os contêineres associativos) localiza um valor especificado em um contêiner.
- As funções `lower_bound` e `upper_bound` (disponíveis em todos os contêineres associativos) localizam a primeira ocorrência do valor especificado em um contêiner e o elemento depois da última ocorrência do valor especificado em um contêiner, respectivamente.

- A função `equal_range` (disponível em todos os contêineres associativos) retorna um `pair` contendo tanto o resultado de uma operação `lower_bound` como o de uma operação `upper_bound`.
- O contêiner associativo `multimap` é utilizado para rápido armazenamento e recuperação de chaves e valores associados (frequentemente chamados de pares chave/valor).
- As chaves duplicadas são permitidas em um `multimap`, portanto múltiplos valores podem ser associados com uma única chave. Isso é chamado relacionamento de um para muitos.
- O arquivo de cabeçalho `<map>` deve ser incluído para utilizar os templates da classe `map` e `multimap`.
- As chaves duplicadas não são permitidas em um `map`, desse modo somente um único valor pode ser associado com cada chave. Isso é chamado mapeamento de um para um.
- Um `map` é comumente chamado de array associativo.
- A STL fornece três adaptadores de contêiner — `stack`, `queue` e `priority_queue`.
- Os adaptadores não são contêineres de primeira classe, porque não fornecem a implementação real de estrutura de dados em que os elementos podem ser armazenados e não suportam iteradores.
- Todos os três templates de classe adaptadora fornecem as funções-membro `push` e `pop` para, respectivamente, inserir e remover um elemento em/de cada estrutura de dados adaptadora de maneira apropriada.
- O template da classe `stack` permite inserções na e exclusões da estrutura de dados subjacente em uma extremidade (comumente referida como uma estrutura de dados último a entrar e primeiro a sair). O arquivo de cabeçalho `<stack>` deve ser incluído para utilizar o template da classe `stack`.
- A função-membro `stack_top` retorna uma referência ao elemento superior do `stack` (implementada chamando a função `back` do contêiner subjacente).
- A função-membro `stack_empty` determina se a `stack` está vazia (implementada chamando a função `empty` do contêiner subjacente).
- A função-membro `stack_size` retorna o número de elementos na `stack` (implementada chamando a função `size` do contêiner subjacente).
- O template da classe `queue` permite inserções na parte de trás da estrutura de dados subjacente e exclusões da parte da frente da estrutura de dados subjacente (comumente referida como uma estrutura de dados primeiro a entrar, primeiro a sair). O arquivo de cabeçalho `<queue>` deve ser incluído para utilizar uma `queue` ou um `priority_queue`.
- A função-membro `queue_front` retorna uma referência ao primeiro elemento na `queue` (implementada chamando a função `front` do contêiner subjacente).
- A função-membro `queue_back` retorna uma referência ao último elemento na `queue` (implementada chamando a função `back` do contêiner subjacente).
- A função-membro `queue_empty` determina se a `queue` está vazia (implementada chamando a função `empty` do contêiner subjacente).
- A função-membro `queue_size` retorna o número de elementos na `queue` (implementada chamando a função `size` do contêiner subjacente).
- O template de classe `priority_queue` fornece funcionalidades que permitem inserções na ordem de classificação na estrutura de dados subjacente e exclusões a partir da frente da estrutura de dados subjacente.
- As operações `priority_queue` comuns são `push`, `pop`, `top`, `empty` e `size`.
- Os algoritmos `fill` e `fill_n` configuram cada elemento em um intervalo de elementos de contêiner como um valor específico.
- Os algoritmos `generate` e `generate_n` utilizam uma função geradora para criar valores para cada elemento em um intervalo de elementos de contêiner.
- O algoritmo `equal` compara a igualdade de duas seqüências de valores.
- O algoritmo `mismatch` compara duas seqüências de valores e retorna um par de iteradores que indicam a localização em cada seqüência dos elementos não-correspondentes.
- O algoritmo `lexicographical_compare` compara o conteúdo de dois arrays de caractere.
- O algoritmo `remove` elimina todos os elementos com um valor específico em certo intervalo.
- O algoritmo `remove_copy` copia todos os elementos que não têm um valor específico em certo intervalo.
- O algoritmo `remove_if` exclui todos os elementos que satisfazem à condição `if` de certo intervalo.
- O algoritmo `remove_copy_if` copia todos os elementos que satisfazem à condição `if` de certo intervalo.
- O algoritmo `replace` substitui todos os elementos por um valor específico de certo intervalo.
- O algoritmo `replace_copy` copia todos os elementos com um valor específico em certo intervalo.
- O algoritmo `replace_if` substitui todos os elementos que satisfazem à condição `if` em certo intervalo.
- O algoritmo `replace_copy_if` copia todos os elementos que satisfazem à condição `if` de certo intervalo.
- O algoritmo `random_shuffle` reordena aleatoriamente os elementos em certo intervalo.

- O algoritmo `count` conta os elementos com um valor específico em certo intervalo.
- O algoritmo `count_if` conta os elementos que satisfazem à condição `if` em certo intervalo.
- O algoritmo `min_element` localiza o menor elemento em certo intervalo.
- O algoritmo `max_element` localiza o maior elemento em certo intervalo.
- O algoritmo `accumulate` soma os valores em certo intervalo.
- O algoritmo `for_each` aplica uma função geral a cada elemento em certo intervalo.
- O algoritmo `transform` aplica uma função geral a cada elemento em certo intervalo e substitui cada elemento pelo resultado da função.
- O algoritmo `find` localiza um valor específico em certo intervalo.
- O algoritmo `find_if` localiza o primeiro valor em certo intervalo que satisfaça a condição `if`.
- O algoritmo `sort` organiza os elementos em certo intervalo na ordem crescente ou em uma ordem especificada por um predicado.
- O algoritmo `binary_search` determina se um valor específico está em certo intervalo.
- O algoritmo `swap` permuta dois valores.
- O algoritmo `iter_swap` permuta dois elementos.
- O algoritmo `swap_ranges` permuta os elementos em certo intervalo.
- O algoritmo `copy_backward` copia elementos em certo intervalo e os coloca em resultados de trás para a frente.
- O algoritmo `merge` combina duas seqüências classificadas crescentes de valores em uma terceira seqüência classificada crescente.
- O algoritmo `unique` remove elementos duplicados em uma seqüência classificada de elementos em certo intervalo.
- O algoritmo `reverse` inverte todos os elementos em certo intervalo.
- O algoritmo `inplace_merge` mescla duas seqüências classificadas de elementos no mesmo contêiner.
- O algoritmo `unique_copy` faz uma cópia de todos os elementos únicos na seqüência classificada de valores em certo intervalo.
- O algoritmo `reverse_copy` faz uma cópia invertida dos elementos em certo intervalo.
- A função `set_includes` compara dois `sets` de valores classificados para determinar se cada elemento do segundo `set` está no primeiro `set`.
- A função `set_set_difference` localiza os elementos do primeiro `set` de valores classificados que não estão no segundo `set` de valores classificados (ambos os `sets` de valores devem estar em ordem crescente).
- A função `set_set_intersection` determina os elementos do primeiro `set` de valores classificados que estão no segundo `set` de valores classificados (ambos os `sets` de valores devem estar em ordem crescente).
- A função `set_set_symmetric_difference` determina os elementos no primeiro `set` que não estão no segundo `set` e os elementos no segundo `set` que não estão no primeiro `set` (ambos os `sets` de valores devem estar em ordem crescente).
- A função `set_set_union` cria um `set` de todos os elementos que estão em qualquer um ou nos dois `sets` classificados (ambos os `sets` de valores devem estar em ordem crescente).
- O algoritmo `lower_bound` encontra a primeira localização em uma seqüência classificada de valores em que o terceiro argumento poderia ser inserido na seqüência de tal modo que a seqüência ainda estaria classificada em ordem crescente.
- O algoritmo `upper_bound` encontra a última localização em uma seqüência classificada de valores em que o terceiro argumento poderia ser inserido na seqüência de tal maneira que a seqüência ainda estaria classificada em ordem crescente.
- O algoritmo `make_heap` aceita uma seqüência de valores em certo intervalo e cria um `heap` que pode ser utilizado para produzir uma seqüência classificada.
- O algoritmo `sort_heap` classifica uma seqüência de valores em certo intervalo, valores estes que já estão organizados em um `heap`.
- O algoritmo `pop_heap` remove o elemento superior do `heap`.
- Os algoritmos `min` e `max` determinam o mínimo e o máximo de dois elementos, respectivamente.
- O template da classe `bitset` facilita a criação e a manipulação de conjuntos de bits que são úteis para representar um conjunto de flags de bits.
- Um objeto de função é uma instância de uma classe que sobrecarrega `operator()`.
- A STL fornece muitos objetos de função predefinidos, que podem ser localizados no cabeçalho `<functional>`.
- Os objetos de função binários são objetos de função que aceitam dois argumentos e retornam um valor. O template da classe `binary_function` é uma classe básica vazia para criar objetos de função binários.

## Terminologia

<algorithm>, arquivo de cabeçalho	flip, função de bitset	priority_queue, template de classe adaptadora
<deque>, arquivo de cabeçalho	for_each, algoritmo	push, função-membro de adaptadores de contêiner
<functional>, arquivo de cabeçalho	front, função-membro de contêiner de sequência	push_heap, algoritmo
<list>, arquivo de cabeçalho	front_inserter, template de função função binária	random_shuffle, algoritmo
<map>, arquivo de cabeçalho	generate, algoritmo	random-access, iterador de acesso aleatório ()
<numeric>, arquivo de cabeçalho	generate_n, algoritmo	rbegin, função-membro de vector
<queue>, arquivo de cabeçalho	heap	remove, algoritmo
<set>, arquivo de cabeçalho	heapsort, algoritmo de classificação	remove, função-membro de list
<stack>, arquivo de cabeçalho	includes, algoritmo	remove_copy, algoritmo
accumulate, algoritmo	inplace_merge, algoritmo	remove_copy_if, algoritmo
adaptador	insert, função-membro de contêineres	remove_if, algoritmo
adaptador de contêiner	inserter, template de função	rend, função-membro de contêineres
algoritmo	intervalo	replace, algoritmo
algoritmo modificador de sequência	istream_iterator	replace_copy, algoritmo
array associativo	iter_swap, algoritmo	replace_copy_if, algoritmo
assign, função-membro de list	iterador	replace_if, algoritmo
back, função-membro de contêineres de sequência	iterador bidirecional	reset, função de bitset
back_inserter, template de função	iterador de entrada	reverse, algoritmo
begin, função-membro, de contêineres de primeira classe	iterador de leitura direta ( <i>forward</i> )	reverse_copy, algoritmo
binary_function, template de classe	iterador de saída	reverse_iterator,
binary_search, algoritmo,	less<int>	second, membro de dados de pair
capacity, função-membro de vector	lexicographical_compare, algoritmo	semicontêiner
chave de pesquisa	list, contêiner de sequência	sequência
chave/valor, par	lower_bound, algoritmo	sequência de entrada
const_iterator	lower_bound, função de contêiner associativo	sequência de saída
const_reverse_iterator	make_heap, algoritmo	set, contêiner associativo
contêiner	map, contêiner associativo	set_difference, algoritmo
contêiner associativo	max, algoritmo	set_intersection, algoritmo
contêiner de primeira classe	max_element, algoritmo	set_symmetric_difference, algoritmo
contêiner de sequência	merge, algoritmo	set_union, algoritmo
copy_backward, algoritmo	min, algoritmo	size, função-membro de contêineres
count, algoritmo	min_element, algoritmo	sort, algoritmo
count_if, algoritmo	mismatch, algoritmo	sort, função-membro de list
deque, contêiner de sequência	multimap, contêiner associativo	sort_heap, algoritmo
empty, função-membro de contêineres	multiset, contêiner associativo	splice, função-membro de list
end, função-membro de contêineres	objeto de função	Standard Template Library (STL)
equal, algoritmo	objeto de função binário	swap, algoritmo
equal_range, algoritmo	objeto de função comparador	swap, função-membro de list
equal_range, função de contêiner associativo	ostream_iterator,	swap_range, algoritmo
erase, função-membro de contêineres	pop, função-membro de adaptadores de contêiner	top, função-membro de adaptadores de contêiner
fill, algoritmo	pop_back, função	um para um, mapeamento
fill_n, algoritmo	pop_front, função	unique, algoritmo
find, algoritmo	pop_heap, algoritmo	unique, função-membro de list
find, função de contêiner associativo	priority_queue, template de classe adaptadora	unique_copy, algoritmo
find_if, algoritmo		upper_bound, algoritmo
first, membro de dados de pair		



## Exercícios de revisão

Determine se as seguintes sentenças são *verdadeiras* ou *falsas* ou preencha as lacunas. Se a resposta for *falsa*, explique por quê.

- 23.1 (V/F) A STL faz uso abundante de herança e funções virtual.
- 23.2 Os dois tipos de contêineres STL são contêineres de sequência e contêineres \_\_\_\_\_.
- 23.3 Os cinco principais tipos de iterador são \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_ e \_\_\_\_\_.
- 23.4 (V/F) Um iterador atua como um ponteiro para um elemento.
- 23.5 (V/F) Os algoritmos STL podem operar em arrays baseados em ponteiros no estilo C.
- 23.6 (V/F) Os algoritmos STL são encapsulados como funções-membro dentro de cada classe contêiner.
- 23.7 (V/F) O algoritmo `remove` não diminui o tamanho do vector a partir do qual os elementos estão sendo removidos.
- 23.8 Os três adaptadores de contêiner STL são \_\_\_\_\_, \_\_\_\_\_ e \_\_\_\_\_.
- 23.9 (V/F) A função-membro de contêiner `end` produz a posição do último elemento do contêiner.
- 23.10 Os algoritmos STL operam em elementos de contêiner indiretamente, utilizando \_\_\_\_\_.
- 23.11 O algoritmo `sort` requer um iterador \_\_\_\_\_.

## Respostas dos exercícios de revisão

- 23.1 Falsa. Esses foram evitados por razões de desempenho.
- 23.2 Associativos.
- 23.3 Entrada, saída, leitura direta (*forward*), bidirecional e acesso aleatório.
- 23.4 Falsa. Na realidade, é o contrário (vice-versa).
- 23.5 Verdadeira.
- 23.6 Falsa. Os algoritmos STL não são funções-membro. Eles operam indiretamente em contêineres, por meio de iteradores.
- 23.7 Verdadeira.
- 23.8 `stack`, `queue`, `priority_queue`.
- 23.9 Falsa. Na realidade, ela fornece a posição imediatamente depois do fim do contêiner.
- 23.10 Iteradores.
- 23.11 De acesso aleatório.

## Exercícios

- 23.12 Escreva um template de função `palindrome` que aceita um parâmetro `vector` e retorna `true` se `vector` lê a mesma coisa da esquerda para a direita e da direita para a esquerda (por exemplo, um `vector` contendo 1, 2, 3, 2, 1 é um palíndromo, mas um `vector` contendo 1, 2, 3, 4 não é).
- 23.13 Modifique a Figura 23.40, o Crivo de Eratóstenes, para que, se o número que o usuário insere no programa não for primo, o programa exiba os fatores primos do número. Lembre-se de que os fatores de um número primo são somente 1 e o próprio número primo. Cada número que não é primo tem uma fatoração em primos única. Por exemplo, os fatores de 54 são 2, 3, 3 e 3. Quando esses valores são multiplicados, o resultado é 54. Para o número 54, a saída dos fatores primos deve ser 2 e 3.
- 23.14 Modifique o Exercício 23.13 para que, se o número que o usuário insere no programa não for primo, o programa exiba os fatores primos do número e quantas vezes cada fator primo aparece na fatoração em primos única. Por exemplo, a saída do número 54 deve ser

The unique prime factorization of 54 is: 2 \* 3 \* 3 \* 3

## Leitura recomendada

- Ammeraal, L. *STL for C++ programmers*. Nova York: John Wiley, 1997.
- Austern, M. H. *Generic programming and the STL: using and extending the C++ Standard Template Library*. Reading, MA: Addison-Wesley, 1998.
- Glass, G., and B. Schuchert. *The STL <Primer>*. Upper Saddle River, NJ: Prentice Hall PTR, 1995.
- Henricson, M., and E. Nyquist. *Industrial strength C++: rules and recommendations*. Upper Saddle River, NJ: Prentice Hall, 1997.
- Josuttis, N. *The C++ Standard Library: a tutorial and handbook*. Reading, MA: Addison-Wesley, 1999.

- Koenig, A., and B. Moo. *Ruminations on C++*. Reading, MA: Addison-Wesley, 1997.
- Meyers, S. *Effective STL: 50 specific ways to improve your use of the Standard Template Library*. Reading, MA: Addison-Wesley, 2001.
- Musser, D. R., and A. Saini. *STL tutorial and reference guide: C++ programming with the Standard Template Library*. Reading, MA: Addison-Wesley, 1996.
- Musser, D. R., and A. A. Stepanov. “Algorithm-oriented generic libraries”, *Software practice and experience*, Vol. 24, Nº 7, julho de 1994.
- Nelson, M. *C++ Programmer’s guide to the Standard Template Library*. Foster City, CA: Programmer’s Press, 1995.
- Pohl, I. *C++ Distilled: a concise ANSI/ISO reference and style guide*. Reading, MA: Addison-Wesley, 1997.
- Pohl, I. *Object-oriented programming using C++, second edition*. Reading, MA: Addison-Wesley, 1997.
- Robson, R. *Using the STL: the C++ Standard Template Library*. Nova York: Springer Verlag, 2000.
- Schildt, H. *STL Programming from the ground up*. Nova York: Osborne McGraw-Hill, 1999.
- Stepanov, A., and M. Lee. “The Standard Template Library”, *Internet distribution* 31 de outubro de 1995 <[www.cs.rpi.edu/~musser/doc.ps](http://www.cs.rpi.edu/~musser/doc.ps)>.
- Stroustrup, B. “Making a vector fit for a standard”, *The C++ report*, outubro de 1994.
- Stroustrup, B. *The design and evolution of C++*. Reading, MA: Addison-Wesley, 1994.
- Stroustrup, B. *The C++ programming language, third edition*. Reading, MA: Addison-Wesley, 1997.
- Vilot, M. J. “An introduction to the Standard Template Library”, *The C++ Report*, Vol. 6, Nº 8, outubro de 1994.



*Que há num simples nome? O  
que chamamos rosa, sob uma  
outra designação teria igual  
perfume.*

William Shakespeare

*O Diamond!<sup>1</sup> Diamond! thou  
little knowest the mischief done!  
[Ah, Diamante! Diamante! Você  
não tem idéia do estrago que  
fez.]*

Sir Isaac Newton

## Outros tópicos

### OBJETIVOS

Neste capítulo, você aprenderá:

- A utilizar `const_cast` para tratar temporariamente um objeto `const` como um objeto não-`const`.
- A utilizar namespaces.
- A utilizar palavras-chave de operador.
- A utilizar membros `mutable` em objetos `const`s.
- A utilizar operadores de ponteiro de membro de classe `.*` e `->*`.
- A utilizar herança múltipla.
- Qual é o papel das classes `virtual` básicas na herança múltipla.

<sup>1</sup> Nome do cão de estimação de Isaac Newton, que derrubou uma vela acesa sobre os trabalhos do cientista.

- 24.1 Introdução
- 24.2 Operador `const_cast`
- 24.3 namespaces
- 24.4 Palavras-chave de operador
- 24.5 Membros de classe `mutable`
- 24.6 Ponteiros para membros de classe (`.*` e `->*`)
- 24.7 Herança múltipla
- 24.8 Herança múltipla e classes básicas `virtual`
- 24.9 Síntese
- 24.10 Observações finais

Resumo | Terminologia | Exercícios de revisão | Respostas dos exercícios de revisão | Exercícios

## 24.1 Introdução

Agora vamos considerar vários recursos C++ avançados. Primeiro você aprenderá sobre o operador `const_cast`, que permite aos programadores adicionar ou remover a qualificação `const` de uma variável. Em seguida, discutimos os namespaces, que podem ser utilizados para assegurar que cada identificador em um programa tenha um nome único e possa ajudar a resolver os conflitos de atribuição de nome causados pelo uso de bibliotecas que têm variáveis, funções ou classes com o mesmo nome. Então apresentamos várias palavras-chave de operador que são úteis para programadores cujos teclados não suportam certos caracteres utilizados em símbolos de operador, como `!`, `&`, `^`, `~` e `|`. Continuamos nossa discussão com o especificador de classe de armazenamento `mutable`, que permite ao programador indicar que um membro de dados deve ser sempre modificável, mesmo quando ele aparecer em um objeto que esteja sendo atualmente tratado como um objeto `const` pelo programa. Em seguida, introduzimos dois operadores especiais que podem ser utilizados com ponteiros para membros de classe a fim de acessar um membro de dados ou função-membro sem saber seu nome antecipadamente. Por fim, introduzimos a herança múltipla, que permitir que uma classe derivada herde o membro de várias classes básicas. Como parte dessa introdução, discutimos problemas potenciais com a herança múltipla e a maneira como a herança `virtual` pode ser utilizada para resolver esses problemas.

## 24.2 Operador `const_cast`

O C++ fornece o operador `const_cast` para fazer coerção da qualificação `const` ou `volatile`. Um programa declara uma variável com o qualificador `volatile` quando esse programa espera que a variável seja modificada por hardware ou por outros programas não conhecidos pelo compilador. Declarar uma variável `volatile` indica que o compilador não deve otimizar o uso dessa variável porque fazer isso poderia afetar a capacidade de acesso de outros programas e modificar a variável `volatile`.

Em geral, é perigoso utilizar o operador `const_cast`, porque ele permite que um programa modifique uma variável que foi declarada `const` e que, portanto, não deve ser modificável. Há casos em que é desejável, ou mesmo necessário, fazer coerção `const`. Por exemplo, bibliotecas C e C++ mais antigas talvez forneçam funções com parâmetros não-`const` e que não modificam seus parâmetros. Se você quisesse passar dados `const` para tal função, precisaria fazer coerção `const` dos dados; caso contrário, o compilador informaria mensagens de erro.

De maneira semelhante, você poderia passar os dados não-`const` para uma função que os trata como se eles fossem constantes e, então, retorna esses dados como uma constante. Em casos assim, talvez você precise fazer coerção `const` dos dados retornados, como demonstramos na Figura 24.1.

Nesse programa, a função `maximum` (linhas 11–14) recebe duas strings no estilo C como parâmetros `const char *` e retorna um `const char *` que aponta para a maior das duas strings. A função `main` declara as duas strings no estilo C como arrays não-`const char` (linhas 18–19); portanto, esses arrays são modificáveis. Em `main`, desejamos gerar saída da maior das duas strings no estilo C e então modificar essa string no estilo C convertendo-a em letras maiúsculas.

Os dois parâmetros da função `maximum` são do tipo `const char *`, então o tipo de retorno da função também deve ser declarado como `const char *`. Se o tipo de retorno é especificado apenas como `char *`, o compilador emite uma mensagem de erro indicando que o valor sendo retornado não pode ser convertido de `const char *` para `char *` — uma conversão perigosa, porque tenta tratar dados que a função acredita ser `const` como se eles fossem dados não-`const`.

Mesmo que a função `maximum` acredite que os dados são constantes, sabemos que os arrays originais em `main` não contêm dados constantes. Portanto, `main` deve ser capaz de modificar o conteúdo desses arrays conforme necessário. Visto que sabemos que esses arrays são modificáveis, utilizamos `const_cast` (linha 23) para fazer coerção `const` do ponteiro retornado por `maximum`, portanto podemos modificar os dados no array que representa a maior das duas strings no estilo C. Podemos então utilizar o ponteiro como o nome de um array de caracteres na instrução `for` (linhas 27–28) para converter o conteúdo da string maior em letras maiúsculas. Sem o `const_cast` na linha 23, esse programa não compilará, porque você não tem permissão de atribuir um ponteiro do tipo `const char *` a um ponteiro do tipo `char *`.

```

1 // Figura 24.1: fig24_01.cpp
2 // Demonstrando const_cast.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <cstring> // contém protótipos para as funções strcmp e strlen
8 #include <cctype> // contém o protótipo para a função toupper
9
10 // retorna a maior entre duas strings no estilo C
11 const char *maximum(const char *first, const char *second)
12 {
13 return (strcmp(first, second) >= 0 ? first : second);
14 } // fim da função maximum
15
16 int main()
17 {
18 char s1[] = "hello"; // array modificável de caracteres
19 char s2[] = "goodbye"; // array modificável de caracteres
20
21 // const_cast requerido para permitir que const char * retornado por máximo
22 // seja atribuído à variável char * maxPtr
23 char *maxPtr = const_cast< char * >(maximum(s1, s2));
24
25 cout << "The larger string is: " << maxPtr << endl;
26
27 for (size_t i = 0; i < strlen(maxPtr); i++)
28 maxPtr[i] = toupper(maxPtr[i]);
29
30 cout << "The larger string capitalized is: " << maxPtr << endl;
31 return 0;
32 } // fim de main

```

```

The larger string is: hello
The larger string capitalized is: HELLO

```

**Figura 24.1** Demonstrando o operador `const_cast`.



### Dica de prevenção de erro 24.1

*Em geral, um `const_cast` só deve ser utilizado quando já se sabe que os dados originais não são constantes. Caso contrário, podem ocorrer resultados inesperados.*

## 24.3 namespaces

Um programa inclui muitos identificadores definidos em diferentes escopos. Às vezes uma variável de um escopo irá se ‘sobrepor’ a (isto é, colidir com) uma variável de mesmo nome em um escopo diferente, criando um possível conflito de atribuição de nome. Tal sobreposição pode ocorrer em muitos níveis. A sobreposição de identificadores costuma ocorrer em bibliotecas independentes que venham a utilizar os mesmos nomes para identificadores globais (como funções). Isso pode causar erros de compilador.



### Boa prática de programação 24.1

*Evite identificadores que iniciam com o caractere de sublinhado, o qual pode levar a erros de linker. Muitas bibliotecas de código utilizam nomes que iniciam com sublinhados.*

O padrão C++ tenta resolver esse problema com **namespaces**. Todo namespace define um escopo em que identificadores e variáveis são colocados. Para utilizar um **membro de namespace**, o nome do membro deve ser qualificado com o nome namespace e com o operador de resolução de escopo binário (`::`), como em

`MeuNameSpace::membro`

ou uma declaração `using` ou diretiva `using` deve aparecer antes de o nome ser utilizado no programa. Em geral, tais instruções `using` são colocadas no início do arquivo em que membros do namespace são utilizados. Por exemplo, colocar a seguinte diretiva `using` no início de um arquivo de código-fonte

```
using namespace MeuNameSpace;
```

especifica que os membros do namespace `MeuNameSpace` podem ser utilizados no arquivo sem preceder cada membro com `MeuNameSpace` e o operador de resolução de escopo (`::`).

Uma declaração `using` (por exemplo, `using std::cout;`) traz um nome no escopo em que a declaração aparece. Uma diretiva `using` (por exemplo, `using namespace std;`) traz todos os nomes do namespace especificado no escopo em que a diretiva aparece.



### Observação de engenharia de software 24.1

*Idealmente, em programas grandes, toda entidade deve ser declarada em uma classe, função, bloco ou namespace. Isso ajuda a esclarecer o papel de cada entidade.*



### Dica de prevenção de erro 24.2

*Preceda um membro com seu nome namespace e com o operador de resolução de escopo (`::`) se existir a possibilidade de um conflito de atribuição de nome.*

Nem todos os namespaces são garantidamente únicos. Dois fornecedores independentes poderiam usar inadvertidamente os mesmos identificadores para seus nomes de namespace. A Figura 24.2 demonstra o uso de namespaces.

#### Utilizando o namespace `std`

A linha 4 informa ao compilador que o namespace `std` está sendo utilizado. O conteúdo do arquivo de cabeçalho `<iostream>` é inteiramente definido como parte do namespace `std`. [Nota: A maioria dos programadores em C++ considera uma prática pobre escrever uma diretiva `using`, como a linha 4, porque todo o conteúdo do namespace está incluído, aumentando assim a probabilidade de um conflito de atribuição de nome.]

A diretiva `using namespace` especifica que os membros de um namespace serão utilizados freqüentemente por todo um programa. Isso permite ao programador acessar todos os membros do namespace e escrever instruções mais concisas como

```
cout << "double1 = " << double1;
```

em vez de

```
std::cout << "double1 = " << double1;
```

Sem a linha 4, cada `cout` e `endl` na Figura 24.2 teria de ser qualificado com `std::`, ou declarações `using` individuais precisariam ser incluídas em `cout` e `endl` como em:

```
using std::cout;
using std::endl;
```

A diretiva `using namespace` pode ser utilizada para namespaces predefinidos (por exemplo, `std`) ou namespaces definidos pelo programador.

#### Definindo `NameSpaces`

As linhas 9–24 utilizam a palavra-chave `namespace` para definir o namespace `Example`. O corpo de um namespace é delimitado por chaves (`{}`). Os membros do namespace `Example` consistem em duas constantes (`PI` e `E` nas linhas 12–13), um `int` (`integer1` na linha 14), uma função (`printValues` na linha 16) e um **namespace aninhado** (`Inner` nas linhas 19–23). Note que o membro `integer1` tem o mesmo nome da variável global `integer1` (linha 6). As variáveis que têm o mesmo nome devem ter escopos diferentes — caso contrário, ocorrem erros de compilação. Um namespace pode conter dados constantes, classes, namespaces aninhados, funções etc. As definições de um namespace devem ocupar o escopo global ou ser aninhadas dentro de outros namespaces.

As linhas 27–30 criam um **namespace não identificado** contendo o membro `doubleInUnnamed`. O namespace não identificado tem uma diretiva `using` implícita, desse modo seus membros parecem ocupar o **namespace global**, são acessíveis diretamente e não têm de ser qualificados com um nome de namespace. As variáveis globais também fazem parte do namespace global e são acessíveis em todos os escopos que se seguem à declaração no arquivo.



### Observação de engenharia de software 24.2

*Cada unidade de compilação separada tem seu próprio namespace não identificado único; isto é, o namespace não identificado substitui o especificador de linkagem `static`.*

```

1 // Figura 24.2: fig24_02.cpp
2 // Demonstrando namespaces.
3 #include <iostream>
4 using namespace std; // utiliza o namespace std
5
6 int integer1 = 98; // variável global
7
8 // cria o namespace Example
9 namespace Example
10 {
11 // declara duas constantes e uma variável
12 const double PI = 3.14159;
13 const double E = 2.71828;
14 int integer1 = 8;
15
16 void printValues(); // protótipo
17
18 // namespace aninhado
19 namespace Inner
20 {
21 // define a enumeração
22 enum Years { FISCAL1 = 1990, FISCAL2, FISCAL3 };
23 } // fim do namespace Inner
24 } // fim do namespace Example
25
26 // cria um namespace não identificado
27 namespace
28 {
29 double doubleInUnnamed = 88.22; // declara a variável
30 } // fim do namespace não identificado
31
32 int main()
33 {
34 // gera saída do valor doubleInUnnamed do namespace não identificado
35 cout << "doubleInUnnamed = " << doubleInUnnamed;
36
37 // gera saída da variável global
38 cout << "\n(global) integer1 = " << integer1;
39
40 // gera saída dos valores do namespace Example
41 cout << "\nPI = " << Example::PI << "\nE = " << Example::E
42 << "\ninteger1 = " << Example::integer1 << "\nFISCAL3 = "
43 << Example::Inner::FISCAL3 << endl;
44
45 Example::printValues(); // invoca a função printValues
46 return 0;
47 } // fim do main
48
49 // exibe os valores variáveis e constantes
50 void Example::printValues()
51 {
52 cout << "\nIn printValues:\ninteger1 = " << integer1 << "\nPI = "
53 << PI << "\nE = " << E << "\ndoubleInUnnamed = "
54 << doubleInUnnamed << "\n(global) integer1 = " << ::integer1
55 << "\nFISCAL3 = " << Inner::FISCAL3 << endl;
56 } // fim do printValues

```

**Figura 24.2** Demonstrando o uso de namespaces.

(continua)



```
doubleInUnnamed = 88.22
(global) integer1 = 98
PI = 3.14159
E = 2.71828
integer1 = 8
FISCAL3 = 1992

In printValues:
integer1 = 8
PI = 3.14159
E = 2.71828
doubleInUnnamed = 88.22
(global) integer1 = 98
FISCAL3 = 1992
```

**Figura 24.2** Demonstrando o uso de namespaces.

(continuação)

### *Acessando membros de namespace com nomes qualificados*

A linha 35 gera saída do valor de variável `doubleInUnnamed`, que é diretamente acessível como parte do namespace não identificado. A linha 38 gera saída do valor da variável global `integer1`. Para essas duas variáveis, o compilador tenta primeiro localizar uma declaração local das variáveis em `main`. Visto que não há declarações locais, o compilador pressupõe que essas variáveis estão no namespace global.

As linhas 41–43 geram saída dos valores de `PI`, `E`, `integer1` e `FISCAL3` a partir do namespace `Example`. Note que cada um deles deve ser qualificado com `Example::` porque o programa não fornece nenhuma diretiva ou declarações `using` que indiquem que ele utilizará membros de namespace `Example`. Além disso, o membro `integer1` deve ser qualificado, porque uma variável global tem o mesmo nome. Caso contrário, a saída do valor da variável global é gerada. Note que `FISCAL3` é o membro do namespace aninhado `Inner`, portanto deve ser qualificado com `Example::Inner::`.

A função `printValues` (definida nas linhas 50–56) é membro de `Example`, portanto ela pode acessar outros membros do namespace `Example` diretamente sem utilizar um qualificador de namespace. A instrução de saída nas linhas 52–55 gera saída `integer1`, `PI`, `E`, `doubleInUnnamed`, variáveis globais `integer1` e `FISCAL3`. Note que `PI` e `E` não são qualificados com `Example`. A variável `doubleInUnnamed` ainda é acessível, porque está no namespace não identificado e o nome variável não entra em conflito com nenhum outro membro de namespace `Example`. A versão global de `integer1` deve ser qualificada com o operador unário de resolução de escopo (`::`), porque seu nome entra em conflito com um membro de namespace `Example`. Além disso, `FISCAL3` deve ser qualificado com `Inner::`. Ao acessar os membros de um namespace aninhado, os membros devem ser qualificados com o nome de namespace (a menos que o membro esteja sendo utilizado dentro do namespace aninhado).



## **Erro comum de programação 24.1**

*Colocar `main` em um namespace é um erro de compilação.*

### *Apelidos para nomes de namespaces*

Namespaces podem ter apelidos. Por exemplo, a instrução

```
namespace CPPHTP5E = CPlusPlusHowToProgram5E;
```

cria o apelido `CPPHTP5E` para `CPlusPlusHowToProgram5E`.

## **24.4 Palavras-chave de operador**

O padrão C++ fornece as **palavras-chave de operador** (Figura 24.3) que podem ser utilizadas no lugar dos vários operadores C++. As palavras-chave de operador são úteis para programadores cujos teclados não suportam certos caracteres como `!`, `&`, `^`, `~`, `|` etc.

A Figura 24.4 demonstra as palavras-chave de operador. Esse programa foi compilado com o Microsoft Visual C++ .NET, que requer que o arquivo de cabeçalho `<iso646.h>` (linha 8) utilize as palavras-chave de operador. No GNU C++, a linha 8 deve ser removida e o programa deve ser compilado como mostrado a seguir:

```
g++ -foperator-names Fig24_04.cpp -o Fig24_04
```

A opção de compilador `-foperator-names` indica que o compilador deve permitir o uso das palavras-chave de operador da Figura 24.3. Outros compiladores não podem requerer que você inclua um arquivo de cabeçalho ou utilize uma opção de compilador para permitir suporte para essas palavras-chave. Por exemplo, o compilador da Borland C++ 5.6.4 permite essas palavras-chave implicitamente.

Operador	Palavra-chave de operador	Descrição
<i>Palavras-chave de operador lógico</i>		
&&	<code>and</code>	E lógico
	<code>or</code>	OU lógico
!	<code>not</code>	NÃO lógico
<i>Palavra-chave de operador de desigualdade</i>		
!=	<code>not_eq</code>	desigualdade
<i>Palavras-chave de operador de bits</i>		
&	<code>bitand</code>	E sobre bits
	<code>bitor</code>	OU inclusivo sobre bits
^	<code>xor</code>	OU exclusivo sobre bits
~	<code>compl</code>	complemento de bits
<i>Palavras-chave de operador de atribuição de bits</i>		
&=	<code>and_eq</code>	atribuição E sobre bits
=	<code>or_eq</code>	atribuição OU inclusivo sobre bits
^=	<code>xor_eq</code>	atribuição OU exclusivo sobre bits

**Figura 24.3** Alternativas de palavra-chave de operador aos símbolos de operador.

```

1 // Figura 24.4: fig24_04.cpp
2 // Demonstrando as palavras-chave de operador.
3 #include <iostream>
4 using std::boolalpha;
5 using std::cout;
6 using std::endl;
7
8 #include <iso646.h> // permite palavras-chave de operador no Microsoft Visual C++
9
10 int main()
11 {
12 bool a = true;
13 bool b = false;
14 int c = 2;
15 int d = 3;
16
17 // a configuração aderente que faz com que valores bool apareçam como true ou false
18 cout << boolalpha;
19
20 cout << "a = " << a << "; b = " << b
21 << "; c = " << c << "; d = " << d;
22
23 cout << "\n\nLogical operator keywords:";
24 cout << "\n a and a: " << (a and a);
25 cout << "\n a and b: " << (a and b);
26 cout << "\n a or a: " << (a or a);

```

**Figura 24.4** Demonstrando as palavras-chave de operador.

(continua)

```

27 cout << "\n a or b: " << (a or b);
28 cout << "\n not a: " << (not a);
29 cout << "\n not b: " << (not b);
30 cout << "\na not_eq b: " << (a not_eq b);
31
32 cout << "\n\nBitwise operator keywords: ";
33 cout << "\nc bitand d: " << (c bitand d);
34 cout << "\nc bit_or d: " << (c bitor d);
35 cout << "\n c xor d: " << (c xor d);
36 cout << "\n compl c: " << (compl c);
37 cout << "\nc and_eq d: " << (c and_eq d);
38 cout << "\n c or_eq d: " << (c or_eq d);
39 cout << "\nc xor_eq d: " << (c xor_eq d) << endl;
40 return 0;
41 } // fim do main

```

```
a = true; b = false; c = 2; d = 3
```

Logical operator keywords:

```

a and a: true
a and b: false
a or a: true
a or b: true
not a: false
not b: true
a not_eq b: true

```

Bitwise operator keywords:

```

c bitand d: 2
c bit_or d: 3
 c xor d: 1
 compl c: -3
c and_eq d: 2
c or_eq d: 3
c xor_eq d: 0

```

**Figura 24.4** Demonstrando as palavras-chave de operador.

(continuação)

O programa declara e inicializa duas variáveis `bool` e duas variáveis do tipo inteiro (linhas 12–15). As operações lógicas (linhas 24–30) são realizadas com as variáveis `bool` `a` e `b` utilizando as várias palavras-chave de operador lógico. As operações de bits (linhas 33–39) são realizadas com as variáveis `int` `c` e `d` utilizando as várias palavras-chave de operador de bits. A saída do resultado de cada operação é gerada.

## 24.5 Membros de classe mutable

Na Seção 24.2, introduzimos o operador `const_cast`, que permitiu remover o tipo `const`. Uma operação `const_cast` também pode ser aplicada a um membro de dados de um objeto `const` a partir do corpo de uma função-membro `const` da classe desse objeto. Isso permite à função-membro `const` modificar o membro de dados, mesmo que o objeto seja considerado `const` no corpo dessa função. Tal operação pode ser realizada quando a maioria dos membros de dados de um objeto deve ser considerada `const`, mas um membro particular de dados ainda precisa ser modificado.

Como um exemplo, considere uma lista vinculada que mantém seu conteúdo em ordem classificada. Pesquisar pela lista vinculada não requer modificações nos dados da lista vinculada, portanto a função de pesquisa poderia ser uma função-membro `const` da classe de listas vinculadas. Entretanto, é concebível que um objeto lista vinculada, em um esforço para tornar as pesquisas futuras mais eficientes, pudesse monitorar a localização da última correspondência bem-sucedida. Se a próxima operação de pesquisa tentar localizar um item que aparece mais adiante na lista, a pesquisa poderia iniciar a partir da localização da última correspondência bem-sucedida, em vez de desde o início da lista. Para fazer isso, a função-membro `const` que realiza a pesquisa deve ser capaz de modificar o membro de dados que monitora a última pesquisa bem-sucedida.

Se um membro de dados como o descrito deve ser sempre modificável, o C++ fornece o especificador de classe de armazenamento `mutable` como alternativa a `const_cast`. Um membro de dados `mutable` é sempre modificável, mesmo em uma função-membro `const` ou objeto `const`. Isso reduz a necessidade de fazer coerção de ‘const’.



### Dica de portabilidade 24.1

*O efeito de tentar modificar um objeto que foi definido como constante, independentemente de essa modificação ter sido ou não possibilitada por um `const_cast` ou por uma coerção no estilo C, varia entre compiladores.*

Tanto `mutable` como `const_cast` permitem que um membro de dados seja modificado; eles são utilizados em contextos diferentes. Para um objeto `const` sem membros de dados `mutable`, o operador `const_cast` deve ser utilizado toda vez que um membro deve ser modificado. Isso reduz significativamente a chance de um membro ser modificado de modo acidental, porque o membro não é permanentemente modificável. As operações que envolvem `const_cast` são, em geral, ocultadas na implementação de uma função-membro. O usuário de uma classe talvez não esteja ciente de que membro está sendo modificado.



### Observação de engenharia de software 24.3

*Os membros `mutable` são úteis em classes que têm detalhes ‘secrets’ de implementação que não contribuem para o valor lógico de um objeto.*

#### Demonstração mecânica de um membro de dados `mutable`

A Figura 24.5 demonstra a utilização de um membro `mutable`. O programa define a classe `TestMutable` (linhas 8–22), que contém um construtor, uma função `getValue` e um membro de dados `private value` que é declarado `mutable`. As linhas 16–19 definem a função `getValue` como uma função-membro `const` que retorna uma cópia de `value`. Note que a função incrementa o membro de dados `mutable value` na instrução `return`. Normalmente, uma função-membro `const` não pode modificar membros de dados a menos que o objeto em que a função opera — por exemplo, aquele para o qual `this` aponta — sofra coerção (utilizando `const_cast`) para um tipo não-`const`. Como `value` é `mutable`, essa função `const` é capaz de modificar os dados.

A linha 26 declara o objeto `const TestMutable test` e o inicializa como 99. A linha 28 chama a função-membro `const getValue`, que adiciona um a `value` e retorna seu conteúdo anterior. Note que o compilador permite a chamada à função-membro `getValue` no objeto `test` porque este é um objeto `const` e `getValue` é uma função-membro `const`. Entretanto, `getValue` modifica a variável `value`. Portanto, quando a linha 29 invoca `getValue` novamente, o novo `value` (100) é enviado para a saída a fim de provar que o membro de dados `mutable` foi de fato modificado.

```

1 // Figura 24.5: fig24_05.cpp
2 // Demonstrando o especificador de classe de armazenamento mutable.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 // definição da classe TestMutable
8 class TestMutable
9 {
10 public:
11 TestMutable(int v = 0)
12 {
13 value = v;
14 } // fim do construtor TestMutable
15
16 int getValue() const
17 {
18 return value++; // incrementa value
19 } // fim da função getValue
20 private:
21 mutable int value; // membro mutable
22 }; // fim da classe TestMutable
23

```

Figura 24.5 Demonstrando um membro de dados `mutable`.

(continua)

```

24 int main()
25 {
26 const TestMutable test(99);
27
28 cout << "Initial value: " << test.getValue();
29 cout << "\nModified value: " << test.getValue() << endl;
30 return 0;
31 } // fim de main

```

```

Initial value: 99
Modified value: 100

```

**Figura 24.5** Demonstrando um membro de dados mutable.

(continuação)

## 24.6 Ponteiros para membros de classe (. \* e ->\*)

O C++ fornece os operadores `. *` e `-> *` para acessar membros de classe via ponteiros. Essa é uma capacidade raramente empregada que é usada principalmente por programadores em C++ avançados. Fornecemos apenas um exemplo mecânico do uso de ponteiros para membros de classe aqui. A Figura 24.6 demonstra os operadores de ponteiro para membro de classe.

O programa declara a classe `Test` (linhas 8–17), que fornece a função-membro `public test` e o membro de dados `public value`. As linhas 19–20 fornecem os protótipos para as funções `arrowStar` (definida nas linhas 32–36) e `dotStar` (definida nas linhas 39–43), que demonstram os operadores `-> *` e `. *`, respectivamente. A linha 24 cria o objeto `test`, e a linha 25 atribui 8 a seu membro de dados `value`. As linhas 26–27 chamam as funções `arrowStar` e `dotStar` com o endereço do objeto `test`.

A linha 34 na função `arrowStar` declara e inicializa a variável `memPtr` como um ponteiro para uma função-membro. Nessa declaração, `Test::*` indica que a variável `memPtr` é um ponteiro para o membro da classe `Test`. Para declarar um ponteiro para uma função, coloque o nome de ponteiro precedido por `*` entre parênteses, como em `( Test::*memPtr )`. Um ponteiro para uma função deve especificar, como parte de seu tipo, tanto o tipo de retorno da função para o qual ele aponta como a lista de parâmetros dessa função. O tipo de retorno da função aparece à esquerda do parêntese esquerdo e a lista de parâmetros aparece em um conjunto separado de parênteses à direita da declaração de ponteiro. Nesse caso, a função tem um tipo de retorno `void` e nenhum parâmetro. O ponteiro `memPtr` é inicializado com o endereço da função-membro da classe `Test` chamado `test`. Observe que o cabeçalho da função deve corresponder à declaração do ponteiro de função — isto é, a função `test` deve ter um tipo de retorno `void` e nenhum parâmetro. Note que o lado direito da atribuição utiliza o operador endereço (`&`) para obter o endereço da função-membro `test`. Além disso, note que nem o lado esquerdo nem o lado direito da atribuição na linha 34 referencia um objeto específico da classe `Test`. Apenas o nome de classe é utilizado com o operador de resolução de escopo binário (`::`). A linha 35 invoca a função-membro armazenada em `memPtr` (isto é, `test`), utilizando o operador `-> *`. Como `memPtr` é um ponteiro para o membro de uma classe, o operador `-> *` deve ser utilizado em vez de o operador `->` invocar a função.

A linha 41 declara e inicializa `vPtr` como um ponteiro para um membro de dados `int` da classe `Test`. O lado direito da atribuição especifica o endereço do membro de dados `value`. A linha 42 desreferencia o ponteiro `testPtr2` e, então, utiliza o operador `. *` para acessar o membro para o qual `vPtr` aponta. Observe que o código de cliente pode criar ponteiros para membros de classe apenas para aqueles membros de classe que são acessíveis pelo código de cliente. Nesse exemplo, tanto a função-membro `test` como o membro de dados `value` são publicamente acessíveis.



### Erro comum de programação 24.2

*Declarar um ponteiro de função-membro sem colocar o nome de ponteiro entre parênteses é um erro de sintaxe.*



### Erro comum de programação 24.3

*Declarar um ponteiro de função-membro sem preceder o nome de ponteiro com um nome de classe seguido pelo operador de resolução de escopo (`::`) é um erro de sintaxe.*



### Erro comum de programação 24.4

*Tentar utilizar o operador `->` ou `*` com um ponteiro para um membro de classe gera erros de sintaxe.*

```

1 // Figura 24.6: fig24_06.cpp
2 // Demonstrando os operadores .* e ->*.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 // definição da classe Test
8 class Test
9 {
10 public:
11 void test()
12 {
13 cout << "In test function\n";
14 } // fim da função test
15
16 int value; // membro de dados public
17 }; // fim da classe Test
18
19 void arrowStar(Test *); // protótipo
20 void dotStar(Test *); // protótipo
21
22 int main()
23 {
24 Test test;
25 test.value = 8; // atribui o valor 8
26 arrowStar(&test); // passa o endereço para arrowStar
27 dotStar(&test); // passa o endereço para dotStar
28 return 0;
29 } // fim do main
30
31 // acessa função-membro do objeto Test utilizando ->*
32 void arrowStar(Test *testPtr)
33 {
34 void (Test::*memPtr)() = &Test::test; // declara o ponteiro de função
35 (testPtr->*memPtr)(); // invoca a função indiretamente
36 } // fim do arrowStar
37
38 // acessa membros do membro de dados do objeto Test utilizando .*
39 void dotStar(Test *testPtr2)
40 {
41 int Test::*vPtr = &Test::value; // declara o ponteiro
42 cout << (*testPtr2).*vPtr << endl; // acessa o valor
43 } // fim do dotStar

```

```

In test function
8

```

**Figura 24.6** Demonstrando os operadores .\* e ->\*.

## 24.7 Herança múltipla

Nos capítulos 9 e 10, discutimos a herança única, em que cada classe é derivada exatamente de uma classe básica. Em C++, uma classe pode ser derivada de mais de uma classe básica — uma técnica conhecida como **herança múltipla**, em que uma classe derivada herda os membros de duas ou mais classes básicas. Essa poderosa capacidade encoraja formas interessantes de reutilização de software, mas pode causar uma variedade de problemas de ambigüidade. A herança múltipla é um conceito difícil que deve ser utilizado somente por programadores experientes. De fato, alguns dos problemas associados com a herança múltipla são tão sutis que linguagens de programação mais novas, como Java e C#, não permitem que uma classe derive de mais de uma classe básica.

**Boa prática de programação 24.2**

*A herança múltipla é uma capacidade poderosa quando usada adequadamente. A herança múltipla deve ser utilizada quando existir um relacionamento ‘é um’ entre um novo tipo e dois ou mais tipos existentes (isto é, o tipo A ‘é um’ tipo B e o tipo A ‘é um’ tipo C).*

**Observação de engenharia de software 24.4**

*A herança múltipla pode introduzir complexidade em um sistema. É necessário muito cuidado no design de um sistema para utilizar a herança múltipla adequadamente; ela não deve ser utilizada quando a herança simples e/ou composição podem fazer o trabalho.*

Um problema comum com a herança múltipla é que cada uma das classes básicas talvez contenham membros de dados ou funções-membro com o mesmo nome. Isso pode levar a problemas de ambigüidade quando você tentar compilar. Considere o exemplo de múltipla herança (figuras 24.7, 24.8, 24.9, 24.10 e 24.11). A classe Base1 (Figura 24.7) contém um membro de dados `protected int` — `value` (linha 20), um construtor (linhas 10–13) que configura `value` e a função-membro `public getData` (linhas 15–18) que retorna `value`.

A classe Base2 (Figura 24.8) é semelhante à classe Base1, exceto pelo fato de que seus dados `protected` são um `char` chamado `letter` (linha 20). Como a classe Base1, a Base2 tem uma função-membro `public getData`, mas essa função retorna o valor de membro de dados `char letter`.

A classe Derived (figuras 24.9–24.10) herda tanto da classe Base1 como da classe Base2 por herança múltipla. A classe Derived tem um membro de dados `private` do tipo `double` chamado `real` (linha 21), um construtor para inicializar todos os dados da classe Derived e uma função-membro `public getReal` que retorna o valor da variável `double real`.

Note como é simples e direto indicar a herança múltipla colocando-se depois do caractere de dois-pontos (`:`) que se segue a `class Derived` uma lista separada por vírgulas de classes básicas (linha 14). Na Figura 24.10, note que o construtor Derived chama explicitamente os construtores de classe básica para cada uma de suas classes básicas — Base1 e Base2 — utilizando a sintaxe inicializadora de membros (linha 9). Os construtores de classe básica são chamados na ordem em que a herança é especificada, não na ordem em que seus construtores são mencionados; além disso, se os construtores de classe básica não forem explicitamente chamados na lista inicializadora de membros, seus construtores-padrão serão chamados implicitamente.

O operador de inserção de fluxo sobrecarregado (Figura 24.10, linhas 18–23) utiliza seu segundo parâmetro — uma referência a um objeto Derived — para exibir os dados de um objeto Derived. Essa função operadora é `friend` de Derived, então `operator<<` pode acessar diretamente todos os membros `protected` e `private` da classe Derived, incluindo o membro de dados `protected value` (herdado da classe Base1), o membro de dados `protected letter` (herdado da classe Base2) e o membro de dados `private real` (declarado na classe Derived).

```

1 // Figura 24.7: Base1.h
2 // Definição da classe Base1
3 #ifndef BASE1_H
4 #define BASE1_H
5
6 // definição da classe Base1
7 class Base1
8 {
9 public:
10 Base1(int parameterValue)
11 {
12 value = parameterValue;
13 } // fim do construtor Base1
14
15 int getData() const
16 {
17 return value;
18 } // fim da função getData
19 protected: // acessíveis a classes derivadas
20 int value; // herdada pela classe derivada
21 }; // fim da classe Base1
22
23 #endif // BASE1_H

```

**Figura 24.7** Demonstrando herança múltipla — Base1.h.



```

1 // Figura 24.8: Base2.h
2 // Definição da classe Base2
3 #ifndef BASE2_H
4 #define BASE2_H
5
6 // definição da classe Base2
7 class Base2
8 {
9 public:
10 Base2(char characterData)
11 {
12 letter = characterData;
13 } // fim do construtor Base2
14
15 char getData() const
16 {
17 return letter;
18 } // end function getData
19 protected: // acessíveis a classes derivadas
20 char letter; // herdada pela classe derivada
21 }; // fim da classe Base2
22
23 #endif // BASE2_H

```

**Figura 24.8** Demonstrando herança múltipla — Base2.h.

```

1 // Figura 24.9: Derived.h
2 // Definição de classe Derived que herda
3 // múltiplas classes básicas (Base1 e Base2).
4 #ifndef DERIVED_H
5 #define DERIVED_H
6
7 #include <iostream>
8 using std::ostream;
9
10 #include "Base1.h"
11 #include "Base2.h"
12
13 // definição da classe Derived
14 class Derived : public Base1, public Base2
15 {
16 friend ostream &operator<<(ostream &, const Derived &);
17 public:
18 Derived(int, char, double);
19 double getReal() const;
20 private:
21 double real; // dados private da classe derivada
22 }; // fim da classe Derived
23
24 #endif // DERIVED_H

```

**Figura 24.9** Demonstrando herança múltipla — Derived.h.

```

1 // Figura 24.10: Derived.cpp
2 // Definições de função-membro para a classe Derived
3 #include "Derived.h"
4
5 // construtor para construtores de chamadas Derived para
6 // as classes Base1 e Base2.
7 // utiliza inicializadores de membro para chamar construtores de classe básica
8 Derived::Derived(int integer, char character, double double1)
9 : Base1(integer), Base2(character), real(double1) { }
10
11 // retorna real
12 double Derived::getReal() const
13 {
14 return real;
15 } // fim da função getReal
16
17 // exibe todos os membros de dados de Derived
18 ostream &operator<<(ostream &output, const Derived &derived)
19 {
20 output << " Integer: " << derived.value << "\n Character: "
21 << derived.letter << "\nReal number: " << derived.real;
22 return output; // permite chamadas em cascata
23 } // fim do operator<<

```

**Figura 24.10** Demonstrando herança múltipla — Derived.cpp.

Agora vamos examinar a função main (Figura 24.11) que testa as classes nas figuras 24.7–24.10. A linha 13 cria o objeto Base1 base1 e o inicializa como o valor int 10 e, então, cria o ponteiro base1Ptr e o inicializa como um ponteiro nulo (isto é, 0). A linha 14 cria o objeto Base2 base2 e o inicializa com o valor char 'Z', e então cria o ponteiro base2Ptr e o inicializa como um ponteiro nulo. A linha 15 cria o objeto Derived derived e o inicializa para conter o valor int 7, o valor char 'A' e o valor double 3.5.

As linhas 18–20 exibem os valores dos dados de cada objeto. Para os objetos base1 e base2, invocamos função-membro getData de cada objeto. Mesmo que haja duas funções getData nesse exemplo, as chamadas não são ambíguas. Na linha 18, o compilador sabe que base1 é um objeto da classe Base1, então a versão da classe Base1 de getData é chamada. Na linha 19, o compilador sabe que base2 é um objeto da classe Base2, então a versão da classe Base2 de getData é chamada. A linha 20 exibe o conteúdo do objeto derived utilizando o operador de inserção de fluxo sobrecarregado.

*Resolvendo questões de ambigüidade que surgem quando uma classe derivada herda funções-membro do mesmo nome de múltiplas classes básicas*

As linhas 24–27 geram saída do conteúdo do objeto derived utilizando novamente as funções-membro get da classe Derived. Entretanto, há um problema de ambigüidade, porque esse objeto contém duas funções getData, uma herdada da classe Base1 e, a outra, da classe Base2. Esse problema é fácil de resolver utilizando o operador de resolução de escopo binário. A expressão derived.Base1::getData() obtém o valor da variável herdada da classe Base1 (isto é, a variável int identificada como value) e derived.Base2::getData() obtém o valor da variável herdada da classe Base2 (isto é, a variável char identificada como letter). O valor double em real é impresso sem ambigüidade com a chamada derived.getReal() — não há nenhuma outra função-membro com esse nome na hierarquia.

*Demonstrando os relacionamentos é um na herança múltipla*

Os relacionamentos *é um* também se aplicam em relacionamentos de múltipla herança. Para demonstrar isso, a linha 31 atribui o endereço de objeto derived ao ponteiro Base1 base1Ptr. Isso é permitido porque um objeto da classe Derived *é um* objeto da classe Base1. A linha 32 invoca a função-membro Base1 getData via base1Ptr para obter apenas o valor da parte Base1 do objeto derived. A linha 35 atribui o endereço do objeto derived ao ponteiro Base2 base2Ptr. Isso é permitido porque um objeto da classe Derived *é um* objeto da classe Base2. A linha 36 invoca a função-membro Base2 getData via base2Ptr para obter apenas o valor da parte Base2 do objeto derived.

## 24.8 Herança múltipla e classes básicas virtual

Na Seção 24.7, discutimos a herança múltipla, o processo pelo qual a classe herda de duas ou mais classes. A herança múltipla é utilizada, por exemplo, na biblioteca C++ padrão para formar a classe basic\_iostream (Figura 24.12).

```

1 // Figura 24.11: fig24_11.cpp
2 // Driver para o exemplo de herança múltipla.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Base1.h"
8 #include "Base2.h"
9 #include "Derived.h"
10
11 int main()
12 {
13 Base1 base1(10), *base1Ptr = 0; // cria objeto Base1
14 Base2 base2('Z'), *base2Ptr = 0; // cria objeto Base2
15 Derived derived(7, 'A', 3.5); // cria objeto Derived
16
17 // imprime membros de dados de objetos da classe básica
18 cout << "Object base1 contains integer " << base1.getData()
19 << "\nObject base2 contains character " << base2.getData()
20 << "\nObject derived contains:\n" << derived << "\n\n";
21
22 // imprime membros de dados do objeto da classe derivada
23 // o operador de resolução de escopo resolve a ambigüidade de getData
24 cout << "Data members of Derived can be accessed individually:"
25 << "\n Integer: " << derived.Base1::getData()
26 << "\n Character: " << derived.Base2::getData()
27 << "\nReal number: " << derived.getReal() << "\n\n";
28 cout << "Derived can be treated as an object of either base class:\n";
29
30 // trata Derived como um objeto Base1
31 base1Ptr = &derived;
32 cout << "base1Ptr->getData() yields " << base1Ptr->getData() << '\n';
33
34 // trata Derived como um objeto Base2
35 base2Ptr = &derived;
36 cout << "base2Ptr->getData() yields " << base2Ptr->getData() << endl;
37 return 0;
38 } // fim do main

```

```

Object base1 contains integer 10
Object base2 contains character Z
Object derived contains:
 Integer: 7
 Character: A
 Real number: 3.5

```

```

Data members of Derived can be accessed individually:
 Integer: 7
 Character: A
 Real number: 3.5

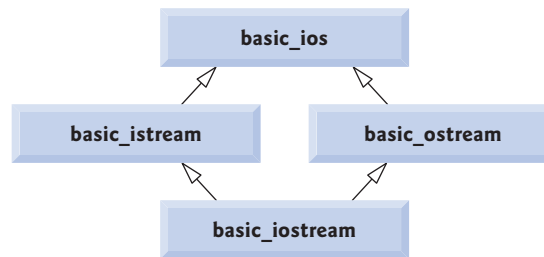
```

```

Derived can be treated as an object of either base class:
base1Ptr->getData() yields 7
base2Ptr->getData() yields A

```

**Figura 24.11** Demonstrando a herança múltipla.



**Figura 24.12** Herança múltipla para formar a classe `basic_iostream`.

A classe `basic_ios` é a classe básica tanto para `basic_istream` como para `basic_ostream`, cada uma das quais formada com a herança simples. A classe `basic_iostream` herda tanto de `basic_istream` como de `basic_ostream`. Isso permite aos objetos da classe `basic_iostream` fornecer as funcionalidades tanto de `basic_istreams` como de `basic_ostreams`. Em hierarquias de múltipla herança, a situação descrita na Figura 24.12 é referida como **herança em forma de losango** (*diamond inheritance*).

Como as classes `basic_istream` e `basic_ostream` herdam cada uma de `basic_ios`, há um problema potencial para `basic_iostream`. A classe `basic_iostream` poderia conter duas cópias dos membros de classe `basic_ios` — uma herdada via classe `basic_istream` e, outra, via classe `basic_ostream`. Tal situação seria ambígua e resultaria em um erro de compilação, porque o compilador não saberia que versão dos membros da classe `basic_ios` utilizar. Naturalmente, `basic_iostream` não sofre do problema que mencionamos. Nesta seção, você verá como utilizar as classes `virtual` básicas para resolver o problema de herdar cópias duplicadas de uma classe básica indireta.

### *Erros de compilação produzidos quando surge ambigüidade na herança em forma de losango*

A Figura 24.13 demonstra a ambigüidade que pode ocorrer na herança em forma de losango. O programa define a classe `Base` (linhas 9–13), que contém a função `virtual print` pura (linha 12). As classes `DerivedOne` (linhas 16–24) e `DerivedTwo` (linhas 27–35) herdam publicamente da classe `Base` e sobreescrevem a função `print`. A classe `DerivedOne` e a classe `DerivedTwo` contêm, cada uma, o que o padrão C++ chama de um **subobjeto de classe básica** — isto é, os membros da classe `Base` nesse exemplo.

A classe `Multiple` (linhas 38–46) herda tanto da classe `DerivedOne` como da classe `DerivedTwo`. Na classe `Multiple`, a função `print` é sobrescrita para chamar `print` de `DerivedTwo` (linha 44). Note que devemos qualificar a chamada `print` com o nome da classe `DerivedTwo` para especificar que versão de `print` chamar.

A função `main` (linhas 48–64) declara objetos das classes `Multiple` (linha 50), `DerivedOne` (linha 51) e `DerivedTwo` (linha 52). A linha 53 declara um array de ponteiros `Base *`. Cada elemento do array é inicializado com o endereço de um objeto (linhas 55–57). Ocorre um erro quando o endereço de `both` — um objeto da classe `Multiple` — é atribuído a `array[ 0 ]`. O objeto `both` realmente contém dois subobjetos do tipo `Base`, então o compilador não sabe para que subobjeto o ponteiro `array[ 0 ]` deve apontar e gera um erro de compilação que indica uma conversão ambígua.

```

1 // Figura 24.13: fig24_13.cpp
2 // Tentando chamar uma função multiplamente
3 // herdada polimorficamente de duas classes básicas.
4 #include <iostream>
5 using std::cout;
6 using std::endl;
7
8 // definição da classe Base
9 class Base
10 {
11 public:
12 virtual void print() const = 0; // virtual pura
13 }; // fim da classe Base
14
15 // definição da classe DerivedOne
16 class DerivedOne : public Base

```

**Figura 24.13** Tentando chamar uma função multiplamente herdada polimorficamente.

(continua)

```

17 {
18 public:
19 // sobrescreve função print
20 void print() const
21 {
22 cout << "DerivedOne\n";
23 } // fim da função print
24 }; // fim da classe DerivedOne
25
26 // definição da classe DerivedTwo
27 class DerivedTwo : public Base
28 {
29 public:
30 // sobrescreve função print
31 void print() const
32 {
33 cout << "DerivedTwo\n";
34 } // fim da função print
35 }; // fim da classe DerivedTwo
36
37 // definição da classe Multiple
38 class Multiple : public DerivedOne, public DerivedTwo
39 {
40 public:
41 // qualifica qual versão da função print
42 void print() const
43 {
44 DerivedTwo::print();
45 } // fim da função print
46 }; // fim da classe Multiple
47
48 int main()
49 {
50 Multiple both; // instancia o objeto Multiple
51 DerivedOne one; // instancia o objeto DerivedOne
52 DerivedTwo two; // instancia objeto DerivedTwo
53 Base *array[3]; // cria array de ponteiros da classe básica
54
55 array[0] = &both; // ERRO--ambíguo
56 array[1] = &one;
57 array[2] = &two;
58
59 // invoca print polimorficamente
60 for (int i = 0; i < 3; i++)
61 array[i] -> print();
62
63 return 0;
64 } // fim de main

```

C:\Projects\cpphttp5\examples\ch24\Fig24\_20\Fig24\_20.cpp(55): error C2594:  
'=' : ambiguous conversions from 'Multiple \*' to 'Base \*'

**Figura 24.13** Tentando chamar uma função multiplamente herdada polimorficamente.

(continuação)

*Eliminando subobjetos duplicados com a herança de classe básica virtual*

O problema de subobjetos duplicados é resolvido com a herança `virtual`. Quando uma classe básica é herdada como `virtual`, somente um subobjeto aparecerá na classe derivada — um processo chamado de **herança da classe básica virtual**. A Figura 24.14 revisa o programa da Figura 24.13 para utilizar uma classe `virtual` básica.

A alteração-chave no programa é que as classes `DerivedOne` (linha 15) e `DerivedTwo` (linha 26) herdam, cada uma, da classe `Base` especificando `virtual public Base`. Como essas duas classes herdam de `Base`, cada uma delas contém um subobjeto `Base`. O benefício da herança `virtual` não é claro até a classe `Multiple` herdar de `DerivedOne` e `DerivedTwo` (linha 37). Visto que cada uma das classes básicas utilizou a herança `virtual` para herdar membros da classe `Base`, o compilador assegura que apenas um subobjeto do tipo `Base` é herdado na classe `Multiple`. Isso elimina o erro de ambigüidade gerado pelo compilador na Figura 24.13. O compilador agora permite a conversão implícita do ponteiro de classe derivada (`&both`) para o ponteiro de classe básica `array[ 0 ]` na linha 56 em `main`. A instrução `for` nas linhas 61–62 chama `print` polimorficamente para cada objeto.

*Construtores em hierarquias de múltipla herança com as classes básicas virtual*

Implementar as hierarquias com as classes básicas `virtual` é mais simples se os construtores-padrão forem utilizados para as classes básicas. Os exemplos nas figuras 24.13 e 24.14 utilizam construtores-padrão gerados por compilador. Se uma classe básica `virtual` fornecer um construtor que requer argumentos, a implementação das classes derivadas torna-se mais complicada, porque a **classe mais derivada** deve invocar explicitamente o construtor da classe básica `virtual` para inicializar os membros herdados da classe básica `virtual`.

**Observação de engenharia de software 24.5**

*Fornecer um construtor-padrão às classes básicas virtual simplifica o design da hierarquia.*

```

1 // Figura 24.14: fig24_14.cpp
2 // Utilizando as classes básicas virtual.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 // definição da classe Base
8 class Base
9 {
10 public:
11 virtual void print() const = 0; // virtual pura
12 }; // fim da classe Base
13
14 // definição da classe DerivedOne
15 class DerivedOne : virtual public Base
16 {
17 public:
18 // sobrescreve a função print
19 void print() const
20 {
21 cout << "DerivedOne\n";
22 } // fim da função print
23 }; // fim da classe DerivedOne
24
25 // definição da classe DerivedTwo
26 class DerivedTwo : virtual public Base
27 {
28 public:
29 // sobrescreve a função print
30 void print() const
31 {
32 cout << "DerivedTwo\n";
33 } // fim da função print

```

**Figura 24.14** Utilizando as classes básicas `virtual`.

(continua)

```

34 }; // fim da classe DerivedTwo
35
36 // definição da classe Multiple
37 class Multiple : public DerivedOne, public DerivedTwo
38 {
39 public:
40 // qualifica qual versão da função print
41 void print() const
42 {
43 DerivedTwo::print();
44 } // fim da função print
45 }; // fim da classe Multiple
46
47 int main()
48 {
49 Multiple both; // instancia o objeto Multiple
50 DerivedOne one; // instancia o objeto DerivedOne
51 DerivedTwo two; // instancia objeto DerivedTwo
52
53 // declara array de ponteiros da classe básica e inicializa
54 // cada elemento para um tipo de classe derivada
55 Base *array[3];
56 array[0] = &both;
57 array[1] = &one;
58 array[2] = &two;
59
60 // invoca polimorficamente a função print
61 for (int i = 0; i < 3; i++)
62 array[i]->print();
63
64 return 0;
65 } // fim do main

```

DerivedTwo  
DerivedOne  
DerivedTwo

**Figura 24.14** Utilizando as classes básicas virtual.

(continuação)

### Informações adicionais sobre a herança múltipla

A herança múltipla é um tópico complexo geralmente abordado em textos C++ mais avançados. Os URLs a seguir fornecem informações adicionais sobre a herança múltipla.

[cplus.about.com/library/weekly/aa121302a.htm](http://cplus.about.com/library/weekly/aa121302a.htm)

Um tutorial sobre herança múltipla com um exemplo detalhado.

[cpptips.hyperformix.com/MultipleInher.html](http://cpptips.hyperformix.com/MultipleInher.html)

Fornecer dicas técnicas que explicam várias questões relacionadas com a herança múltipla.

[www.parashift.com/c++-faq-lite/multiple-inheritance.html](http://www.parashift.com/c++-faq-lite/multiple-inheritance.html)

Parte do C++ *FAQ Lite*. Fornece uma explicação técnica detalhada sobre a herança múltipla e a herança virtual.

## 24.9 Síntese

Neste capítulo, você aprendeu a utilizar o operador `const_cast` para remover a qualificação `const` de uma variável. Então mostramos como utilizar namespaces para assegurar que cada identificador em um programa tenha um nome único e explicado como eles podem ajudar a resolver conflitos de atribuição de nome. Você viu as palavras-chave de operador para programadores cujos teclados não suportam certos caracteres utilizados em símbolos de operador, como `!`, `&`, `^`, `~` e `|`. Em seguida, mostramos como o especificador de classe de armazenamento `mutable` permite ao programador indicar que um membro de dados deve ser sempre modificável, mesmo



quando aparece em um objeto que está sendo atualmente tratado como um `const`. Mostramos também a mecânica do uso de ponteiros para membros de classe e dos operadores `->*` e `.*`. Por fim, introduzimos a herança múltipla e discutimos problemas associados com a permissão de uma classe derivada para herdar o membro de várias classes básicas. Como parte dessa discussão, demonstramos como a herança `virtual` pode ser utilizada para resolver esses problemas.

## 24.10 Observações finais

Esperamos sinceramente que você tenha gostado de aprender C++ e programação orientada a objetos com C++ *Como Programar, 5/e*. Gostaríamos muito de receber seus comentários, críticas, correções e sugestões para melhorar o texto. Envie sua correspondência para nosso endereço de correio eletrônico:

deitel@deitel.com

Boa sorte!

### Resumo

- O C++ fornece o operador `const_cast` para fazer a coerção da qualificação `const` ou `volatile`.
- Um programa declara uma variável com o qualificador `volatile` quando esse programa espera que a variável seja modificada por outros programas. Declarar uma variável `volatile` indica que o compilador não deve otimizar o uso dessa variável porque fazer isso poderia afetar a capacidade de acesso de outros programas e modificar a variável `volatile`.
- Em geral, é perigoso utilizar o operador `const_cast`, porque ele permite que um programa modifique uma variável que foi declarada `const` e que, portanto, não deve ser modificável.
- Há casos em que é desejável, ou mesmo necessário, fazer coerção `const`. Por exemplo, bibliotecas C e C++ mais antigas talvez forneçam funções com parâmetros não-`const` e que não modificam seus parâmetros. Se você quisesse passar dados `const` para tal função, precisaria fazer coerção `const` dos dados; caso contrário, o compilador informaria mensagens de erro.
- Se você passar dados não-`const` para uma função que os trata como se eles fossem uma constante e, então, retorna esses dados como uma constante, você pode precisar fazer coerção `const` dos dados retornados para acessar e modificar esses dados.
- Um programa inclui muitos identificadores definidos em diferentes escopos. Às vezes uma variável de um escopo irá se ‘sobrepor’ a uma variável do mesmo nome em um escopo diferente, criando um provável conflito de atribuição de nome. O C++ padrão tenta resolver esse problema com namespaces.
- Cada namespace define um escopo em que identificadores são colocados. Para utilizar um membro namespace, o nome do membro deve ser qualificado com o nome namespace e o operador de resolução de escopo binário (`:`) ou uma diretiva ou declaração `using` deve aparecer antes de o nome ser utilizado no programa.
- Em geral, as instruções `using` são colocadas no início do arquivo em que membros do namespace são utilizados.
- Nem todos os namespaces são garantidamente únicos. Dois fornecedores independentes poderiam usar inadvertidamente os mesmos identificadores para seus nomes de namespace.
- Uma diretiva `using namespace` especifica que os membros de um namespace serão freqüentemente utilizados por todo um programa. Isso permite ao programador acessar todos os membros do namespace.
- Uma diretiva `using namespace` pode ser utilizada para namespaces predefinidos (por exemplo, `std`) ou namespaces definidos pelo programador.
- Um namespace pode conter dados constantes, classes, namespaces aninhados, funções etc. As definições de namespaces devem ocupar o escopo global ou ser aninhadas dentro de outros namespaces.
- Um namespace não identificado tem uma diretiva `using` implícita, portanto seus membros parecem ocupar o namespace global, são acessíveis diretamente e não têm de ser qualificados com um nome de namespace. As variáveis globais também fazem parte do namespace global.
- Ao acessar os membros de um namespace aninhado, os membros devem ser qualificados com o nome de namespace (a menos que o membro esteja sendo utilizado dentro do namespace aninhado).
- Namespaces podem ter apelidos.
- O C++ padrão fornece as palavras-chave de operador que podem ser utilizadas no lugar de vários operadores C++. As palavras-chave de operador são úteis para programadores cujos teclados não suportam certos caracteres como `!`, `&`, `^`, `~`, `|` etc.
- Se um membro de dados deve ser sempre modificável, o C++ fornece o especificador de classe de armazenamento `mutable` como uma alternativa a `const_cast`. Um membro de dados `mutable` é sempre modificável, mesmo em uma função-membro `const` ou objeto `const`. Isso reduz a necessidade de fazer coerção ‘`const`’.

- Tanto `mutable` como `const_cast` permitem que um membro de dados seja modificado; eles são utilizados em contextos diferentes. Para um objeto `const` sem membros de dados `mutable`, o operador `const_cast` deve ser utilizado toda vez que um membro deve ser modificado. Isso reduz significativamente a chance de um membro ser modificado de modo acidental, porque o membro não é permanentemente modificável.
- As operações que envolvem `const_cast` são, em geral, ocultadas na implementação de uma função-membro. O usuário de uma classe talvez não esteja ciente de que membro está sendo modificado.
- O C++ fornece os operadores `.*` e `->*` para acessar os membros de classe via ponteiros. Essa capacidade raramente utilizada é empregada principalmente por programadores em C++ avançados.
- Declarar um ponteiro para uma função requer que você coloque o nome de ponteiro precedido por um `*` entre parênteses. Um ponteiro para uma função deve especificar, como parte de seu tipo, tanto o tipo de retorno da função para o qual ele aponta como a lista de parâmetros dessa função.
- Em C++, uma classe pode ser derivada de mais de uma classe básica — uma técnica conhecida como herança múltipla, em que uma classe derivada herda os membros de duas ou mais classes básicas.
- Um problema comum com a herança múltipla é que cada uma das classes básicas talvez contenham membros de dados ou funções-membro que tenham o mesmo nome. Isso pode levar a problemas de ambigüidade quando você tentar compilar.
- Os relacionamentos *é um* também se aplicam em relacionamentos de múltipla herança.
- A herança múltipla é utilizada, por exemplo, na C++ Standard Library para formar a classe `basic_iostream`. A classe `basic_ios` é a classe básica tanto para `basic_istream` como para `basic_ostream`, cada uma das quais é formada com a herança simples. A classe `basic_iostream` herda tanto de `basic_istream` como de `basic_ostream`. Isso permite aos objetos da classe `basic_iostream` fornecer as funcionalidades de `basic_istreams` e `basic_ostreams`. Em hierarquias de múltipla herança, a situação descrita aqui é referida como herança em forma de losango.
- Como as classes `basic_istream` e `basic_ostream` herdam cada uma de `basic_ios`, há um problema potencial para `basic_iostream`. Se não implementada corretamente, a classe `basic_iostream` poderia conter duas cópias dos membros da classe `basic_ios` — uma herdada via classe `basic_istream`, e a outra, via classe `basic_ostream`. Tal situação seria ambígua e resultaria em um erro de compilação, porque o compilador não saberia que versão dos membros da classe `basic_ios` utilizar.
- A ambigüidade na herança em forma de losango ocorre quando um objeto de classe derivada herda dois ou mais subobjetos de classe básica. O problema de subobjetos duplicados é resolvido com a herança `virtual`. Quando uma classe básica é herdada como `virtual`, apenas um subobjeto aparecerá na classe derivada — um processo chamado de herança de classe básica `virtual`.
- Implementar as hierarquias com as classes básicas `virtual` é mais simples se os construtores-padrão forem utilizados para as classes básicas. Se uma classe básica `virtual` fornecer um construtor que requer argumentos, a implementação das classes derivadas torna-se mais complicada, porque a **classe mais derivada** deve invocar explicitamente o construtor da classe básica `virtual` para inicializar os membros herdados da classe básica `virtual`.

## Terminologia

<code>.*</code> , operador	lista separada por vírgulas de classes básicas	palavra-chave de operador
<code>-&gt;*</code> , operador	<code>mutable</code> , membro de dados	palavras-chave do operador de atribuição de bits
<code>and</code> , palavra-chave de operador	<code>namespace</code>	palavras-chave do operador de bits
<code>and_eq</code> , palavra-chave de operador	<code>namespace aninhado</code>	palavras-chave do operador de desigualdade
apelido de <code>namespace</code>	<code>namespace global</code>	palavras-chave do operador lógico
<code>bitand</code> , palavra-chave de operador	<code>namespace não-identificado</code>	subobjeto de classe básica
<code>bitor</code> , palavra-chave de operador	<code>namespace</code> , palavra-chave	<code>using</code> , declaração
classe mais derivada	nomes, conflito	<code>using</code> , declaração <code>namespace</code>
coerção <code>const</code>	<code>not</code> , palavra-chave de operador	<code>virtual</code> , classe básica
<code>compl</code> , palavra-chave de operador	<code>not_eq</code> , palavra-chave de operador	<code>virtual</code> , herança
<code>const_cast</code> , operador	operadores de ponteiro para membro	<code>volatile</code> , qualificador
herança em forma de losango	<code>or</code> , palavra-chave de operador	<code>xor</code> , palavra-chave de operador
herança múltipla	<code>or_eq</code> , palavra-chave de operador	<code>xor_eq</code> , palavra-chave de operador

## Exercícios de revisão

**24.1** Preencha as lacunas de cada uma das seguintes sentenças:

- O operador \_\_\_\_\_ qualifica um membro com seu `namespace`.
- O operador \_\_\_\_\_ permite que um objeto sofra a coerção `const`.
- Como um `namespace` não-identificado tem uma diretiva `using` implícita, seus membros parecem ocupar o \_\_\_\_\_, são acessíveis diretamente e não tem de ser qualificados com um nome `namespace`.

- d) O operador `_____` é a palavra-chave de operador para desigualdade.
- e) Uma classe pode ser derivada de mais de uma classe básica; tal derivação é chamada `_____`.
- f) Quando uma classe básica é herdada como `_____`, apenas um subobjeto da classe básica aparecerá na classe derivada.

**24.2** Determine quais das seguintes sentenças são *verdadeiras* e quais são *falsas*. Se for *falsa*, explique por quê.

- a) Ao passar um argumento não-`const` a uma função `const`, o operador `const_cast` deve ser utilizado para fazer coerção '`const`' da função.
- b) Não é garantido que os namespaces sejam únicos.
- c) Semelhantes aos corpos de classe, os corpos namespace também terminam em ponto-e-vírgulas.
- d) Os namespaces não podem ter namespaces como membros.
- e) Um membro de dados `mutable` não pode ser modificado em uma função-membro `const`.

## Respostas dos exercícios de revisão

**24.1** a) resolução de escopo binário (`::`). b) `const_cast`. c) namespace global. d) `not_eq`. e) herança múltipla. f) `virtual`.

**24.2** a) Falsa. É válido passar um argumento não-`const` para uma função `const`. Entretanto, ao passar uma referência ou ponteiro `const` para uma função não-`const`, o operador `const_cast` deve ser utilizado para fazer coerção '`const`' da referência ou do ponteiro.  
 b) Falsa. Os programadores poderiam inadvertidamente escolher o namespace já em uso.  
 c) Falsa. Os corpos namespace não terminam em ponto-e-vírgulas.  
 d) Falsa. Os namespaces podem ser aninhados.  
 e) Falsa. Um membro de dados `mutable` é sempre modificável, mesmo em uma função-membro `const`.

## Exercícios

**24.3** Preencha as lacunas de cada uma das seguintes sentenças:

- a) A palavra-chave `_____` especifica que um namespace ou membro namespace está sendo utilizado.
- b) O operador `_____` é a palavra-chave de operador para OU lógico.
- c) O especificador de armazenamento `_____` permite que um membro de um objeto `const` seja modificado.
- d) O qualificador `_____` especifica que um objeto pode ser modificado por outros programas.
- e) Preceda um membro com seu nome `_____` e o operador de resolução de escopo `_____` se existir a possibilidade de um conflito de escopo.
- f) O corpo de um namespace é delimitado por `_____`.
- g) Para um objeto `const` sem membros de dados `_____`, o operador `_____` deve ser utilizado toda vez que um membro for modificado.

**24.4** Escreva um namespace, `Currency`, que define os membros constantes `ONE`, `TWO`, `FIVE`, `TEN`, `TWENTY`, `FIFTY` e `HUNDRED`. Escreva dois programas curtos que utilizam `Currency`. Um programa deve disponibilizar todas as constantes e o outro deve tornar apenas `FIVE` disponível.

**24.5** Dado os namespaces na Figura 24.15, determine se cada sentença é *verdadeira* ou *falsa*. Explique todas as respostas *falsas*.

- a) A variável `kilometers` é visível dentro do namespace `Data`.
- b) O objeto `string1` é visível dentro do namespace `Data`.
- c) A constante `POLAND` não é visível dentro do namespace `Data`.
- d) A constante `GERMANY` é visível dentro de namespace `Data`.
- e) A função `function` é visível ao namespace `Data`.
- f) O namespace `Data` é visível ao namespace `CountryInformation`.
- g) O objeto `map` é visível ao namespace `CountryInformation`.
- h) O objeto `string1` é visível dentro do namespace `RegionalInformation`.

**24.6** Compare e contraste `mutable` e `const_cast`. Dê pelo menos um exemplo de quando um seria preferível ao outro. [Nota: Este exercício não requer que nenhum código seja escrito.]

**24.7** Escreva um programa que utiliza `const_cast` para modificar uma variável `const`. [Dica: Utilize um ponteiro em sua solução para apontar para o identificador `const`.]

**24.8** Que problema as classes básicas `virtual` resolvem?

**24.9** Escreva um programa que utiliza as classes básicas `virtual`. A classe na parte superior da hierarquia deve fornecer um construtor que aceita pelo menos um argumento (isto é, não fornece um construtor-padrão). Que desafios isso apresenta para a hierarquia de herança?

```

1 namespace CountryInformation
2 {
3 using namespace std;
4 enum Countries { POLAND, SWITZERLAND, GERMANY,
5 AUSTRIA, CZECH_REPUBLIC };
6 int kilometers;
7 string string1;
8
9 namespace RegionalInformation
10 {
11 short getPopulation(); // pressupõe que a definição existe
12 MapData map; // pressupõe que a definição existe
13 } // fim de RegionalInformation
14 } // fim de CountryInformation
15
16 namespace Data
17 {
18 using namespace CountryInformation::RegionalInformation;
19 void *function(void *, int);
20 } // fim de Data

```

**Figura 24.15** Os namespaces para o Exercício 24.5.

**24.10** Localize o(s) erro(s) em cada uma das seguintes instruções. Quando possível, explique como corrigir cada erro.

- a) `namespace Name {`  
`int x;`  
`int y;`  
`mutable int z;`  
`};`
- b) `int integer = const_cast< int >( double );`
- c) `namespace PCM( 111, "hello" ); // constrói namespace`



# Tabela de precedência e associatividade de operadores

## Precedência de operadores

Os operadores são mostrados em ordem de precedência decrescente de cima para baixo (Figura A.1).

Operador	Tipo	Associatividade
:: ::	resolução de escopo binário resolução de escopo unário	da esquerda para a direita
() [] . -> ++ -- typeid dynamic_cast < tipo > static_cast< tipo > reinterpret_cast< tipo > const_cast< tipo >	parênteses subscrito de array seleção de membro via objeto seleção de membro via ponteiro incremento pós-fixado unário decremento pós-fixado unário informações de tipo em tempo de execução coerção com verificação de tipos em tempo de execução coerção com verificação de tipos em tempo de compilação coerção para conversões não-padrão coerção de const-ância	da esquerda para a direita
++ -- + - ! ~ sizeof & *	incremento prefixado unário decremento prefixado unário mais unário menos unário negação lógica unária complemento unário sobre bits determina tamanho em bytes endereço desreferência	da direita para a esquerda

Figura A.1 Tabela de precedência e associatividade de operadores.

(continua)

Operador	Tipo	Associatividade
<code>new</code> <code>new[]</code> <code>delete</code> <code>delete[]</code>	alocação dinâmica de memória alocação dinâmica de array desalocação dinâmica de memória desalocação dinâmica de array	da direita para a esquerda
<code>( tipo )</code>	coerção unária no estilo C	da direita para a esquerda
<code>.*</code> <code>-&gt;*</code>	ponteiro para membro via objeto ponteiro para membro via ponteiro	da esquerda para a direita
<code>*</code> <code>/</code> <code>%</code>	multiplicação divisão módulo	da esquerda para a direita
<code>+</code> <code>-</code>	adição subtração	da esquerda para a direita
<code>&lt;&lt;</code> <code>&gt;&gt;</code>	deslocamento de bits para a esquerda deslocamento de bits para a direita	da esquerda para a direita
<code>&lt;</code> <code>&lt;=</code> <code>&gt;</code> <code>&gt;=</code>	relacional menor que relacional menor que ou igual a relacional maior que relacional maior que ou igual a	da esquerda para a direita
<code>==</code> <code>!=</code>	relacional é igual a relacional não é igual a	da esquerda para a direita
<code>&amp;</code>	E sobre bits	da esquerda para a direita
<code>^</code>	OU exclusivo sobre bits	da esquerda para a direita
<code> </code>	OU inclusivo sobre bits	da esquerda para a direita
<code>&amp;&amp;</code>	E lógico	da esquerda para a direita
<code>  </code>	OU lógico	da esquerda para a direita
<code>?:</code>	ternário condicional	da direita para a esquerda
<code>=</code> <code>+=</code> <code>-=</code> <code>*=</code> <code>/=</code> <code>%=</code> <code>&amp;=</code> <code>^=</code> <code> =</code> <code>&lt;&lt;=</code> <code>&gt;&gt;=</code>	atribuição atribuição de adição atribuição de subtração atribuição de multiplicação atribuição de divisão atribuição de módulo atribuição E sobre bits atribuição OU exclusivo sobre bits atribuição OU inclusivo sobre bits atribuição de deslocamento de bits para a esquerda atribuição de deslocamento de bits para a direita	da direita para a esquerda
<code>,</code>	vírgula	da esquerda para a direita

Figura A.1 Tabela de precedência e associatividade de operadores.

(continuação)

# B

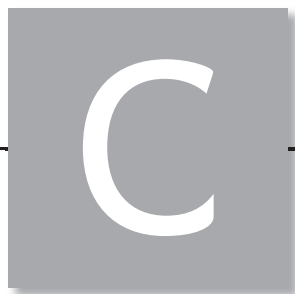
## Conjunto de caracteres ASCII

Conjunto de caracteres ASCII										
	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	lf	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	\$	%	&	'
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	'	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{		}	~	del		

**Figura B.1** Conjunto de caracteres ASCII.

Os dígitos à esquerda da tabela são os dígitos à esquerda do equivalente decimal (0–127) do código de caractere, e os dígitos na parte superior da tabela são os dígitos à direita do código de caractere. Por exemplo, o código de caractere para ‘F’ é 70, e o código de caractere para ‘&’ é 38.





## Tipos fundamentais

A Figura C.1 lista os tipos fundamentais do C++. O C++ Standard Document não fornece o número exato de bytes necessários para armazenar as variáveis desses tipos na memória. Entretanto, o C++ Standard Document indica como os requisitos de memória dos tipos fundamentais se relacionam entre si. Por ordem crescente de requisitos de memória, os tipos inteiros com sinal são `signed char`, `short int`, `int` e `long int`. Isso significa que um `short int` deve fornecer pelo menos a mesma quantidade de armazenamento que um `signed char`; um `int` deve fornecer pelo menos a mesma quantidade de armazenamento que um `short int`; e um `long int` deve fornecer pelo menos a mesma quantidade de armazenamento que um `int`. Cada tipo inteiro com sinal tem um tipo inteiro sem sinal correspondente que possui os mesmos requisitos de memória. Os tipos sem sinal não podem representar valores negativos, mas podem representar duas vezes mais valores positivos do que poderiam seus tipos com sinal associados. Por ordem crescente de requisitos de memória, os tipos de

Tipos integrais	Tipos de ponto flutuante
<code>bool</code>	<code>float</code>
<code>char</code>	<code>double</code>
<code>signed char</code>	<code>long double</code>
<code>unsigned char</code>	
<code>short int</code>	
<code>unsigned short int</code>	
<code>int</code>	
<code>unsigned int</code>	
<code>long int</code>	
<code>unsigned long int</code>	
<code>wchar_t</code>	

**Figura C.1** Tipos fundamentais do C++.

ponto flutuante são `float`, `double` e `long double`. Semelhantemente aos tipos inteiros, um `double` deve fornecer pelo menos a mesma quantidade de armazenamento que um `float`, e um `long double` deve fornecer pelo menos o que um `double`.

O tamanho e o intervalo exato de valores para os tipos fundamentais são dependentes de implementação. Os arquivos de cabeçalho `<climits>` (para os tipos integrais) e `<float>` (para os tipos de ponto flutuante) especificam os intervalos de valores suportados no sistema.

O intervalo de valores que um tipo suporta depende do número de bytes utilizados para representar esse tipo. Por exemplo, considere um sistema com `ints` de 4 bytes (32 bits). Para o tipo `int` com sinal, os valores não-negativos estão no intervalo de 0 a 2.147.483.647 ( $2^{31} - 1$ ). Os valores negativos estão no intervalo de  $-1$  a  $-2.147.483.648$  ( $-2^{31}$ ). Esse é um total de  $2^{32}$  valores possíveis. Um `unsigned int` no mesmo sistema utilizaria o mesmo número de bits para representar os dados, mas não representaria nenhum valor negativo. Isso resulta em valores no intervalo de 0 a 4.294.967.295 ( $2^{32} - 1$ ). No mesmo sistema, um `short int` não poderia utilizar mais de 32 bits para representar seus dados e um `long int` deve utilizar pelo menos 32 bits.

O C++ fornece o tipo de dados `bool` para variáveis que podem armazenar somente os valores `true` e `false`.

# D



*Aqui estão apenas números ratificados.*

William Shakespeare

*A natureza tem algum tipo de sistema de coordenadas geométrico-aritmético, porque a natureza tem todos os tipos de modelos. O que experimentamos de natureza está em modelos e todos os modelos da natureza são belos.*

*Surpreende-me que o sistema da natureza tenha tal beleza, uma vez que em química descobrimos que as associações ocorrem sempre em belos números inteiros — não há nenhuma fração.*

Richard Buckminster Fuller

## Sistemas de numeração

### OBJETIVOS

Neste apêndice, você aprenderá:

- Conceitos básicos de sistemas de numeração como base, valor posicional e valor de símbolo.
- Como trabalhar com números representados nos sistemas de numeração binários, octais e hexadecimais.
- A abreviar números binários como números octais ou números hexadecimais.
- A converter números octais e números hexadecimais em números binários.
- A converter nos dois sentidos entre números decimais e seus equivalentes binários, octais e hexadecimais.
- O que é aritmética binária e como os números binários negativos são representados utilizando a notação de complemento de dois.

- D.1** Introdução
- D.2** Abreviando números binários como números octais e números hexadecimais
- D.3** Convertendo números octais e hexadecimais em números binários
- D.4** Convertendo de octal, binário ou hexadecimal para decimal
- D.5** Convertendo de decimal para octal, binário ou hexadecimal
- D.6** Números binários negativos: notação de complemento de dois

Resumo | Terminologia | Exercícios de revisão | Respostas dos exercícios de revisão | Exercícios

## D.1 Introdução

Neste apêndice introduzimos os principais sistemas de numeração que os programadores em C++ utilizam, especialmente quando estão trabalhando em projetos de software que requerem íntima interação com hardware no nível de máquina. Projetos como esse incluem sistemas operacionais, software de rede de computador, compiladores, sistemas de banco de dados e aplicativos que requerem alto desempenho.

Quando escrevemos um inteiro como 227 ou -63 em um programa C++, o número é entendido como estando no sistema de numeração decimal (base 10). Os dígitos no sistema de numeração decimal são 0, 1, 2, 3, 4, 5, 6, 7, 8 e 9. O dígito mais baixo é 0 e o mais alto é 9 — um a menos que a base 10. Internamente, os computadores utilizam o sistema de numeração binário (base 2). O sistema de numeração binário tem apenas dois dígitos, 0 e 1. Seu dígito mais baixo é 0 e seu mais alto é 1 — um a menos que a base 2.

Como veremos, números binários tendem a ser muito mais longos que seus equivalentes decimais. Os programadores que trabalham em linguagens assembly, e em linguagens de alto nível como C++, que lhes permitem alcançar o nível de máquina, acham incômodo trabalhar com números binários. Então dois outros sistemas de numeração — o sistema de numeração octal (base 8) e o sistema de numeração hexadecimal (base 16) — são populares principalmente porque tornam conveniente abreviar números binários.

No sistema de numeração octal, os dígitos variam de 0 a 7. Como os sistemas de numeração binário e octal têm menos dígitos que o sistema de numeração decimal, seus dígitos são os mesmos que os dígitos correspondentes em decimal.

O sistema de numeração hexadecimal impõe um problema uma vez que requer 16 dígitos — um dígito mais baixo que 0 e um dígito mais alto com um valor equivalente ao decimal 15 (um a menos que a base 16). Por convenção, utilizamos as letras A a F para representar os dígitos hexadecimais correspondentes aos valores decimais de 10 a 15. Portanto, em hexadecimal, podemos ter números como 876 consistindo unicamente em dígitos do tipo decimal, números como 8A55F consistindo em dígitos e letras, e números como FFE consistindo unicamente em letras. Ocasionalmente, um número hexadecimal forma uma palavra comum como FACE ou FOOD — isso pode parecer estranho para programadores acostumados a trabalhar com números. Os dígitos dos sistemas de numeração binário, octal, decimal e hexadecimal são resumidos nas figuras D.1 e D.2.

Cada um desses sistemas de numeração utiliza notação posicional — cada posição na qual um dígito é escrito tem um valor posicional diferente. Por exemplo, no número decimal 937 (o 9, o 3 e o 7 são referenciados como valores de símbolo), dizemos que o 7 é escrito na posição das unidades, o 3 é escrito na posição das dezenas e o 9 é escrito na posição das centenas. Observe que cada uma dessas posições é uma potência da base (base 10) e que essas potências iniciam em 0 e aumentam por 1 à medida que nos movemos para a esquerda no número (Figura D.3).

Para números decimais mais longos, as próximas posições à esquerda seriam a posição dos milhares (10 à terceira potência), a posição das dezenas de milhares (10 à quarta potência), a posição das centenas de milhares (10 à quinta potência), a posição dos milhões (10 à sexta potência), a posição das dezenas de milhões (10 à sétima potência) e assim por diante.

No número binário 101, o 1 mais à direita é escrito na posição das unidades, o 0 é escrito na posição dos 2s e o 1 mais à esquerda é escrito na posição dos 4s. Observe que cada posição é uma potência da base (base 2) e que essas potências iniciam em 0 e aumentam por 1 à medida que nos movemos à esquerda no número (Figura D.4). Portanto,  $101 = 2^2 + 2^0 = 4 + 1 = 5$ .

Para números binários mais longos, as próximas posições à esquerda seriam a posição dos 8s (2 elevado a 3), a posição dos 16s (2 elevado a 4), a posição dos 32s (2 elevado a 5), a posição dos 64s (2 elevado a 6) e assim por diante.

No número octal 425, dizemos que o 5 é escrito na posição das unidades, o 2 é escrito na posição dos 8s e o 4 é escrito na posição dos 64s. Observe que cada uma dessas posições é uma potência da base (base 8) e que essas potências iniciam em 0 e aumentam por 1 quando nos movemos para a esquerda no número (Figura D.5).

Para números octais mais longos, as próximas posições à esquerda seriam a posição dos 512s (8 elevado a 3), a posição dos 4.096s (8 elevado a 4), a posição dos 32.768s (8 elevado a 5) e assim por diante.

No número hexadecimal 3DA, dizemos que A é escrito na posição das unidades, o D é escrito na posição dos 16s e o 3 é escrito na posição dos 256s. Observe que cada uma dessas posições é uma potência da base (base 16) e que essas potências iniciam em 0 e aumentam por 1 à medida que nos movemos para a esquerda no número (Figura D.6).

Para números hexadecimais mais longos, as próximas posições à esquerda seriam a posição dos 4.096(s) (16 elevado a 3), a posição dos 65.536s (16 elevado a 4) e assim por diante.

Dígito binário	Dígito octal	Dígito decimal	Dígito hexadecimal
0	0	0	0
1	1	1	1
	2	2	2
	3	3	3
	4	4	4
	5	5	5
	6	6	6
	7	7	7
		8	8
		9	9
			A (valor decimal de 10)
			B (valor decimal de 11)
			C (valor decimal de 12)
			D (valor decimal de 13)
			E (valor decimal de 14)
			F (valor decimal de 15)

**Figura D.1** Os dígitos dos sistemas de numeração binário, octal decimal e hexadecimal.

Atributo	Binário	Octal	Decimal	Hexadecimal
Base	2	8	10	16
Dígito mais baixo	0	0	0	0
Dígito mais alto	1	7	9	F

**Figura D.2** Comparando os sistemas de numeração binário, octal, decimal e hexadecimal.

Valores posicionais no sistema de numeração decimal			
Dígito decimal	9	3	7
Posicione nome	Centenas	Dezenas	Unidades
Valor posicional	100	10	1
Valor posicional como uma potência da base (10)	$10^2$	$10^1$	$10^0$

**Figura D.3** Valores posicionais no sistema de numeração decimal.

## D.2 Abreviando números binários como números octais e números hexadecimais

A principal utilização para números hexadecimais e octais em computação é abreviar longas representações binárias. A Figura D.7 destaca o fato de que números binários longos podem ser expressos concisamente em sistemas de numeração com bases mais altas que o sistema de numeração binário.

Valores posicionais no sistema de numeração binário			
Dígito binário	1	0	1
Posicione nome	4s	2s	Unidades
Valor posicional	4	2	1
Valor posicional como uma potência da base (2)	$2^2$	$2^1$	$2^0$

**Figura D.4** Valores posicionais no sistema de numeração binário.

Valores posicionais no sistema de numeração octal			
Dígito decimal	4	2	5
Posicione nome	64s	8s	Unidades
Valor posicional	64	8	1
Valor posicional como uma potência da base (8)	$8^2$	$8^1$	$8^0$

**Figura D.5** Valores posicionais no sistema de numeração octal.

Valores posicionais no sistema de numeração hexadecimal			
Dígito decimal	3	D	A
Posicione nome	256s	16s	Unidades
Valor posicional	256	16	1
Valor posicional como uma potência da base (16)	$16^2$	$16^1$	$16^0$

**Figura D.6** Valores posicionais no sistema de numeração hexadecimal.

Um relacionamento particularmente importante que tanto o sistema de numeração octal como o sistema de numeração hexadecimal têm com o sistema binário é que as bases octal e hexadecimal (8 e 16, respectivamente) são potências da base do sistema de numeração binário (base 2). Considere o seguinte número binário de 12 algarismos e seus equivalentes octal e hexadecimal. Veja se você pode determinar como esse relacionamento torna conveniente a abreviação de número binário em octal ou hexadecimal. A resposta segue os números.

Número binário	Equivalente octal	Equivalente hexadecimal
100011010001	4321	8D1

Para ver como o número binário é facilmente convertido em octal, simplesmente divida o número binário de 12 dígitos em grupos de três bits consecutivos cada, iniciando da direita, e escreva esses grupos sobre os dígitos correspondentes do número octal como segue:

100	011	010	001
4	3	2	1

Observe que o dígito octal que você escreveu sob cada grupo de bits corresponde precisamente ao equivalente octal desse número binário de três dígitos, como mostrado na Figura D.7.

O mesmo tipo de relacionamento pode ser observado ao converter de binário para hexadecimal. Quebre o número binário de 12 algarismos em grupos de quatro bits consecutivos cada, iniciando da direita, e escreva esses grupos sobre os dígitos correspondentes do número de hexadecimal, como segue:

1000	1101	0001
8	D	1

Observe que o dígito hexadecimal que você escreveu sob cada grupo de quatro bits corresponde precisamente ao equivalente de hexadecimal do número binário de quatro algarismos, como mostrado na Figura D.7.

Número decimal	Representação binária	Representação octal	Representação hexadecimal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

**Figura D.7** Equivalentes decimais, binários, octais e hexadecimais.

### D.3 Convertendo números octais e hexadecimais em números binários

Na seção anterior, vimos como converter números binários em seus equivalentes octal e hexadecimal formando grupos de dígitos binários e simplesmente reescrevendo-os como seus valores equivalentes de dígito octal ou de dígito hexadecimal. Esse processo pode ser utilizado no sentido inverso para produzir o equivalente binário de um determinado número octal ou hexadecimal.

Por exemplo, o número octal 653 é convertido em binário simplesmente escrevendo o 6 como seu equivalente binário de três dígitos 110, o 5 como seu equivalente binário de três dígitos 101 e o 3 como seu equivalente binário de três dígitos 011 para formar o número binário de nove dígitos 110101011.

O número de hexadecimal FAD5 é convertido em binário simplesmente escrevendo o F como seu equivalente binário de quatro dígitos 1111, o A como seu equivalente binário de quatro dígitos 1010, o D como seu equivalente binário de quatro dígitos 1101 e o 5 como seus equivalentes binários de quatro dígitos 0101 para formar o número de 16 dígitos 1111101011010101.

### D.4 Convertendo de octal, binário ou hexadecimal para decimal

Estamos acostumados a trabalhar no sistema decimal, e, portanto, costuma ser conveniente converter um número octal, binário ou hexadecimal em decimal para obter um sentido de quanto o número “realmente” vale. Nossos diagramas na Seção D.1 expressam os valores posicionais em decimal. Para converter um número em decimal a partir de outra base, multiplique o equivalente decimal de cada dígito por seu valor posicional e some esses produtos. Por exemplo, o número binário 110101 é convertido no decimal 53 como mostra a Figura D.8.

Para converter o octal 7614 no decimal 3980, utilizamos a mesma técnica, dessa vez utilizando valores posicionais octais apropriados, como mostrado na Figura D.9.

Para converter o hexadecimal AD3B no decimal 44347, utilizamos a mesma técnica, dessa vez utilizando valores posicionais hexadecimais apropriados como mostra a Figura D.10.



Convertendo um número binário em decimal						
Valores posicionais:	32	16	8	4	2	1
Valores de símbolo:	1	1	0	1	0	1
Produtos:	$1 \cdot 32 = 32$	$1 \cdot 16 = 16$	$0 \cdot 8 = 0$	$1 \cdot 4 = 4$	$0 \cdot 2 = 0$	$1 \cdot 1 = 1$
Soma:	$= 32 + 16 + 0 + 4 + 0 + 1 = 53$					

Figura D.8 Convertendo um número binário em decimal.

Convertendo um número octal em decimal				
Valores posicionais:	512	64	8	1
Valores de símbolo:	7	6	1	4
Produtos:	$7 \cdot 512 = 3584$	$6 \cdot 64 = 384$	$1 \cdot 8 = 8$	$4 \cdot 1 = 4$
Soma:	$= 3584 + 384 + 8 + 4 = 3980$			

Figura D.9 Convertendo um número octal em decimal.

Convertendo um número hexadecimal em decimal				
Valores posicionais:	4096	256	16	1
Valores de símbolo:	A	D	3	B
Produtos:	$A \cdot 4096 = 40960$	$D \cdot 256 = 3328$	$3 \cdot 16 = 48$	$B \cdot 1 = 11$
Soma:	$= 40960 + 3328 + 48 + 11 = 44347$			

Figura D.10 Convertendo um número hexadecimal em decimal.

## D.5 Convertendo de decimal para octal, binário ou hexadecimal

As conversões na Seção D.4 seguem-se naturalmente das convenções de notação posicional. A conversão de decimal em octal, binário ou hexadecimal também segue essas convenções.

Suponha que quiséssemos converter o decimal 57 em binário. Iniciamos escrevendo os valores posicionais das colunas da direita para a esquerda até alcançarmos uma coluna cujo valor posicional é maior que o número decimal. Não precisamos dessa coluna, então a descartamos. Portanto, primeiro escrevemos:

Valores posicionais:	64	32	16	8	4	2	1
----------------------	----	----	----	---	---	---	---

Então descartamos a coluna com valor posicional 64, deixando:

Valores posicionais:	32	16	8	4	2	1
----------------------	----	----	---	---	---	---

Em seguida, trabalhamos da coluna mais à esquerda para a direita. Dividimos 57 por 32 e observamos que há um 32 em 57 com um resto de 25, então escrevemos 1 na coluna 32. Dividimos 25 por 16 e observamos que há um 16 em 25 com um resto de 9 e escrevemos 1 na coluna 16. Dividimos 9 por 8 e observamos que há um 8 em 9 com um resto de 1. Cada uma das duas próximas colunas produz quocientes de 0 quando 1 é dividido por seus valores posicionais, portanto escrevemos 0s nas colunas 4 e 2. Por fim, 1 em 1 é 1, então escrevemos 1 na coluna 1. Isso resulta em:

Valores posicionais:	32	16	8	4	2	1
Valores de símbolo:	1	1	1	0	0	1

E assim o decimal 57 é equivalente ao binário 111001.

Para converter o decimal 103 em octal, iniciamos escrevendo os valores posicionais das colunas até alcançarmos uma coluna cujo valor posicional é maior que o número decimal. Não precisamos dessa coluna, então a descartamos. Portanto, primeiro escrevemos:

Valores posicionais:	512	64	8	1
----------------------	-----	----	---	---

Então descartamos a coluna com valor posicional 512, o que resulta em:

Valores posicionais:	64	8	1
----------------------	----	---	---

Em seguida trabalhamos da coluna mais à esquerda para a direita. Dividimos 103 por 64 e observamos que há um 64 em 103 com um resto de 39, então escrevemos 1 na coluna 32. Dividimos 39 por 8 e observamos que há quatro 8s em 39 com um resto de 7 e escrevemos 4 na coluna 8. Por fim, dividimos 7 por 1 e observamos que há sete 1s em 7 sem resto, então escrevemos 7 na coluna 1. Isso resulta em:

Valores posicionais:	64	8	1
Valores de símbolo:	1	4	7

e assim o decimal 103 é equivalente ao octal 147.

Para converter o decimal 375 em hexadecimal, iniciamos escrevendo os valores posicionais das colunas até alcançarmos uma coluna cujo valor posicional é maior que o número decimal. Não precisamos dessa coluna, então a descartamos. Portanto, primeiro escrevemos:

Valores posicionais:	4096	256	16	1
----------------------	------	-----	----	---

Então descartamos a coluna com valor posicional 4.096, o que resulta em:

Valores posicionais:	256	16	1
----------------------	-----	----	---

Em seguida trabalhamos da coluna mais à esquerda para a direita. Dividimos 375 por 256 e observamos que há um 256 em 375 com um resto de 119, então escrevemos 1 na coluna 256. Dividimos 119 por 16 e observamos que há sete 16s em 119 com um resto de 7 e escrevemos 7 na coluna 16. Por fim, dividimos 7 por 1 e observamos que há sete 1s em 7 sem resto, então escrevemos 7 na coluna 1. Isso resulta em:

Valores posicionais:	256	16	1
Valores posicionais:	1	7	7

e assim o decimal 375 é equivalente ao hexadecimal 177.

## D.6 Números binários negativos: notação de complemento de dois

A discussão até agora neste apêndice focalizou números positivos. Nesta seção, explicamos como os computadores representam números negativos utilizando notação de complemento de dois. Inicialmente explicamos como o *complemento de dois* de um número binário é formado e então mostramos por que ele representa o valor negativo do número binário dado.

Considere uma máquina com inteiros de 32 bits. Suponha

```
int value = 13;
```

A representação de 32 bits de `value` é

```
00000000 00000000 00000000 00001101
```

Para formar o negativo de `value` primeiro formamos seu *complemento de um* aplicando operador de complemento de bitwise do C++ (~):

```
onesComplementOfValue = ~value;
```

Internamente, `~value` é agora `value` com cada um de seus bits invertidos — 1s tornam-se 0s e 0s tornam-se 1s, como segue:

```
value:
00000000 00000000 00000000 00001101
```

```
~value (isto é, o complemento de um do valor):
11111111 11111111 11111111 11110010
```

Para formar complemento de dois de `value`, simplesmente adicionamos 1 ao complemento de um de `value`. Assim

```
Complemento de dois de value:
11111111 11111111 11111111 11110011
```

Agora, se isso é de fato igual a `-13`, devemos ser capazes de adicioná-lo ao binário 13 e obter um resultado de 0. Vamos tentar isso:

```
00000000 00000000 00000000 00001101
+11111111 11111111 11111111 11110011

00000000 00000000 00000000 00000000
```

O bit de transporte que vem da coluna mais à esquerda é descartado e, de fato, obtemos 0 como resultado. Se adicionarmos o complemento de um número ao número, o resultado será todos 1s. A chave para obter um resultado de todos os dígitos como zeros é que o complemento de dois seja 1 maior que o complemento de um. A adição de 1 faz com que cada coluna adicione 0 a um transportador de 1. O transportador continua se movendo na esquerda até que seja descartado do bit mais à esquerda e daí o número resultante é todos os dígitos como zero.

Os computadores na realidade realizam uma subtração, como

$$x = a - \text{value};$$

adicionando o complemento de dois de `value` a `a`, como segue:

$$x = a + (-\text{value} + 1);$$

Suponha que `a` seja 27 e `value` seja 13 como antes. Se o complemento de dois de `value` é realmente a negativa de `value`, então adicionar o complemento de dois de `value` a `a` deve produzir o resultado 14. Vamos tentar isso:

a (isto é, 27)	00000000 00000000 00000000 00011011
+ (~value + 1)	+11111111 11111111 11111111 11110011
	-----
	00000000 00000000 00000000 00001110

que é de fato igual a 14.

## Resumo

- Um inteiro como 19 ou 227 ou -63 em um programa C++ é entendido como estando no sistema de numeração decimal (base 10). Os dígitos no sistema de numeração decimal são 0, 1, 2, 3, 4, 5, 6, 7, 8 e 9. O dígito mais baixo é 0 e o mais alto é 9 — um a menos que a base 10.
- Internamente, os computadores utilizam o sistema de numeração binário (base 2). O sistema de numeração binário tem apenas dois dígitos, 0 e 1. Seu dígito mais baixo é 0 e seu mais alto é 1 — um a menos que a base 2.
- O sistema de numeração octal (base 8) e o sistema de numeração hexadecimal (base 16) são populares principalmente porque tornam conveniente a abreviação de números binários.
- Os dígitos do sistema de numeração octal variam de 0 a 7.
- O sistema de numeração hexadecimal impõe um problema uma vez que requer 16 dígitos — um dígito mais baixo que 0 e um dígito mais alto com um valor equivalente ao decimal 15 (um a menos que a base 16). Por convenção, utilizamos as letras A a F para representar os dígitos hexadecimais correspondentes aos valores decimais 10 a 15.
- Cada sistema de numeração utiliza notação posicional — cada posição em que um dígito é escrito tem um valor posicional diferente.
- Um relacionamento particularmente importante dos sistemas de numeração octal e hexadecimal com o sistema binário é que suas bases (8 e 16, respectivamente) são potências da base do sistema de numeração binário (base 2).
- Para converter um número octal em um binário, substitua cada dígito octal por seu equivalente binário de três dígitos.
- Para converter um número hexadecimal em um número binário, simplesmente substitua cada dígito hexadecimal por seu equivalente binário de quatro dígitos.
- Uma vez que estamos acostumados a trabalhar no sistema decimal, é conveniente converter um número binário, octal ou hexadecimal em decimal para obter um sentido do valor “real” do número.
- Para converter um número em decimal a partir de outra base, multiplique o equivalente decimal de cada dígito por seu valor posicional e some os produtos.
- Os computadores representam números negativos utilizando a notação de complemento de dois.
- Para formar o negativo de um valor em binário, primeiro forme o complemento de um aplicando o operador de complemento de bits do C++ (~). Isso inverte os bits do valor. Para formar o complemento de dois de um valor, simplesmente adicione 1 ao complemento de um do valor.

## Terminologia

base	operador de complemento de bits ~	sistema de numeração decimal
complemento de dois, notação	sistema de numeração binário	sistema de numeração hexadecimal
conversões	sistema de numeração de base 10	sistema de numeração octal
dígito	sistema de numeração de base 16	valor de símbolo
notação de complemento de um	sistema de numeração de base 2	valor negativo
notação posicional	sistema de numeração de base 8	valor posicional

## Exercícios de revisão

- D.1** As bases dos sistemas de numeração binários, octais, decimais e hexadecimais são \_\_\_\_\_, \_\_\_\_\_, \_\_\_\_\_ e \_\_\_\_\_, respectivamente.
- D.2** Em geral, as representações decimal, octal e hexadecimal de um número binário dado contêm (mais/menos) dígitos que o número binário.

- D.3** (Verdadeiro/Falso) Uma razão popular para utilizar o sistema de numeração decimal é que ele forma uma notação conveniente para abreviar números binários simplesmente substituindo um dígito decimal por grupo de quatro bits binários.
- D.4** A representação (octal / hexadecimal / decimal) de um valor binário grande é a mais concisa (das alternativas dadas).
- D.5** (Verdadeiro/Falso) O dígito mais alto em qualquer base é um maior que a base.
- D.6** (Verdadeiro/Falso) O dígito mais baixo em qualquer base é um menor que a base.
- D.7** O valor posicional do dígito mais à direita de qualquer número em octal, binário, hexadecimal ou decimal é sempre \_\_\_\_\_.
- D.8** O valor posicional do dígito à esquerda do dígito mais à direita de qualquer número em octal binário, hexadecimal ou decimal é sempre igual a \_\_\_\_\_.
- D.9** Preencha os valores ausentes nesse gráfico de valores posicionais para as quatro posições mais à direita em cada um dos sistemas de numeração indicados:
- |             |      |     |     |     |
|-------------|------|-----|-----|-----|
| decimal     | 1000 | 100 | 10  | 1   |
| hexadecimal | ...  | 256 | ... | ... |
| binário     | ...  | ... | ... | ... |
| octal       | 512  | ... | 8   | ... |
- D.10** Converta o binário 110101011000 em octal e hexadecimal.
- D.11** Converta o hexadecimal FACE em binário.
- D.12** Converta o octal 7316 em binário.
- D.13** Converta o octal 4FEC em hexadecimal. [Dica: Primeiro converta 4FEC em binário, e então converta esse número binário em octal.]
- D.14** Converta o binário 1101110 em decimal.
- D.15** Converta o octal 317 em decimal.
- D.16** Converta o hexadecimal EFD4 em decimal.
- D.17** Converta o decimal 177 em binário, em octal e em hexadecimal.
- D.18** Mostre a representação binária do decimal 417. Então, mostre o complemento de um de 417 e o complemento de dois de 417.
- D.19** Qual é o resultado quando um número e seu complemento de dois são somados?

## Respostas dos exercícios de revisão

- D.1** 2, 8, 10, 16.
- D.2** Menos.
- D.3** Falso. O hexadecimal faz isso.
- D.4** Hexadecimal.
- D.5** Falso. O dígito mais alto em qualquer base é um menor que a base.
- D.6** Falso. O dígito mais baixo em qualquer base é zero.
- D.7** 1 (a base elevada à potência zero).
- D.8** A base do sistema de numeração.
- D.9** Preencha os valores ausentes nesse gráfico de valores posicionais para as quatro posições mais à direita em cada um dos sistemas de numeração indicados:
- |             |      |     |    |   |
|-------------|------|-----|----|---|
| decimal     | 1000 | 100 | 10 | 1 |
| hexadecimal | 4096 | 256 | 16 | 1 |
| binário     | 8    | 4   | 2  | 1 |
| octal       | 512  | 64  | 8  | 1 |
- D.10** Octal 6530; hexadecimal D58.
- D.11** Binário 1111 1010 1100 1110.
- D.12** Binário 111 011 001 110.
- D.13** Binário 0 100 111 111 101 100; octal 47754.
- D.14** Decimal  $2+4+8+32+64=$  decimal 110.
- D.15** Decimal  $7+1*8+3*64=7+8+192=207$ .
- D.16** Decimal  $4+13*16+15*256+14*4096=61396$ .

**D.17** Decimal 177

em binário:

256 128 64 32 16 8 4 2 1  
 128 64 32 16 8 4 2 1  
 $(1 \cdot 128) + (0 \cdot 64) + (1 \cdot 32) + (1 \cdot 16) + (0 \cdot 8) + (0 \cdot 4) + (0 \cdot 2) + (1 \cdot 1)$   
 10110001

em octal:

512 64 8 1  
 64 8 1  
 $(2 \cdot 64) + (6 \cdot 8) + (1 \cdot 1)$   
 261

em hexadecimal:

256 16 1  
 16 1  
 $(11 \cdot 16) + (1 \cdot 1)$   
 $(B \cdot 16) + (1 \cdot 1)$   
 B1

**D.18** Binário:

512 256 128 64 32 16 8 4 2 1  
 256 128 64 32 16 8 4 2 1  
 $(1 \cdot 256) + (1 \cdot 128) + (0 \cdot 64) + (1 \cdot 32) + (0 \cdot 16) + (0 \cdot 8) + (0 \cdot 4) + (0 \cdot 2) + (1 \cdot 1)$   
 110100001

Complemento de um: 001011110

Complemento de dois: 001011111

Verificação: Número binário original + seu complemento de dois

110100001  
 001011111  
 -----  
 000000000

**D.19** Zero.**Exercícios**

**D.20** Algumas pessoas argumentam que muitos de nossos cálculos seriam mais fáceis no sistema de numeração de base 12 porque 12 é divisível por muito mais números que 10 (para a base 10). Qual é o dígito mais baixo na base 12? O que seria o símbolo mais alto para o dígito na base 12? Quais são os valores posicionais das quatro posições mais à direita de qualquer número no sistema de numeração de base 12?

**D.21** Complete o seguinte gráfico de valores posicionais para as quatro posições mais à direita em cada um dos sistemas de numeração indicados:

decimal	1000	100	10	1
base 6	...	...	6	...
base 13	...	169	...	...
base 3	27	...	...	...

**D.22** Converta o binário 100101111010 em octal e em hexadecimal.

**D.23** Converta o hexadecimal 3A7D em binário.

**D.24** Converta o hexadecimal 765F em octal. [*Dica:* Primeiro converta 765F em binário, então converta esse número binário em octal.]

**D.25** Converta o binário 1011110 em decimal.

**D.26** Converta o octal 426 em decimal.

**D.27** Converta o hexadecimal FFFF em decimal.

**D.28** Converta o decimal 299 em binário, em octal e em hexadecimal.

**D.29** Mostre a representação binária do decimal 417. Então, mostre o complemento de um de 779 e o complemento de dois de 779.

**D.30** Mostre o complemento de dois do valor inteiro -1 em uma máquina com inteiros de 32 bits.

# E



*Utilizaremos um sinal que eu  
testei e considerei de grande  
alcance e fácil de gritar.  
Uaa-huu!*  
Zane Grey

*É problema que pode ser  
resolvido em três cachimbadas.*  
Sir Arthur Conan Doyle

*Aparentemente separadas, mas  
ainda assim unidas.*  
William Shakespeare

## Tópicos sobre o código C legado

### OBJETIVOS

Neste apêndice, você aprenderá:

- Como redirecionar a entrada pelo teclado para vir de um arquivo e redirecionar a saída na tela para um arquivo.
- A escrever funções que utilizam listas de argumentos de comprimento variável.
- Como processar argumentos de linha de comando.
- Como processar eventos inesperados dentro de um programa.
- Como alocar memória dinamicamente para arrays, utilizando alocação dinâmica de memória no estilo C.
- A redimensionar memória dinamicamente alocada, utilizando alocação dinâmica de memória no estilo C.

- E.1** Introdução
- E.2** Redirecionando entrada/saída em sistemas UNIX/Linux/Mac OS X e Windows
- E.3** Listas de argumentos de comprimento variável
- E.4** Utilizando argumentos de linha de comando
- E.5** Notas sobre a compilação de programas de arquivo de múltiplas fontes
- E.6** Terminação de programa com `exit` e `atexit`
- E.7** O qualificador de tipo `volatile`
- E.8** Sufixos para constantes de inteiro e de ponto flutuante
- E.9** Tratamento de sinal
- E.10** Alocação dinâmica de memória com `calloc` e `realloc`
- E.11** O desvio incondicional: `goto`
- E.12** Uniões
- E.13** Especificações de linkagem
- E.14** Síntese

Resumo | Terminologia | Exercícios de revisão | Respostas dos exercícios de revisão | Exercícios

## E.1 Introdução

Este apêndice apresenta vários tópicos que não são comumente tratados em cursos introdutórios. Muitas das capacidades discutidas aqui são específicas de sistemas operacionais particulares, especialmente UNIX/Linux/Mac OS X e/ou Windows. Grande parte do material é para o benefício de programadores em C++ que precisarão trabalhar com o código C legado mais antigo.

## E.2 Redirecionando entrada/saída em sistemas UNIX/Linux/Mac OS X e Windows

Normalmente, a entrada para um programa é proveniente do teclado (entrada-padrão) e a saída de um programa é exibida na tela (saída-padrão). Na maioria dos sistemas de computador — sistemas UNIX, Linux, Mac OS X e Windows em particular — é possível **redirecionar** entradas de modo que provenham de arquivo e redirecionar saídas para serem localizadas em um arquivo. Ambas as formas de redirecionamento podem ser realizadas sem utilizar as capacidades de processamento de arquivo da biblioteca-padrão.

Há várias maneiras de redirecionar a entrada e a saída a partir da linha de comando UNIX. Considere o arquivo executável `sum` que aceita a entrada de um inteiro por vez, mantém uma soma parcial dos valores até que o indicador de fim do arquivo seja configurado e, então, imprime o resultado. Normalmente o usuário insere inteiros a partir do teclado e insere a combinação de teclas de fim do arquivo para indicar que nenhum outro valor será incluído. Com o redirecionamento de entrada, a entrada pode ser armazenada em um arquivo. Por exemplo, se os dados são armazenados no arquivo `input`, a linha de comando

```
$ sum < input
```

faz com que o programa `sum` seja executado; o **símbolo de redirecionamento de entrada** (`<`) indica que os dados no arquivo `input` (em vez do teclado) devem ser utilizados como entrada pelo programa. O redirecionamento da entrada em um Prompt de comando Windows é realizado de modo idêntico.

Observe que `$` representa o prompt de linha de comando UNIX. (Os prompts UNIX variam de sistema para sistema e entre shells em um único sistema.) O redirecionamento é uma função de sistema operacional, não outro recurso do C++.

O segundo método de redirecionamento de entrada é o **piping**. O caractere de barra vertical ou **pipe** (`|`) faz com que a saída de um programa seja redirecionada como a entrada de outro programa. Suponha que um programa `random` gere a saída de uma série de inteiros aleatórios; a saída de `random` pode ser diretamente redirecionada ('piped') para o programa `sum` utilizando a linha de comando UNIX

```
$ random | sum
```

Isso faz com que a soma dos inteiros produzidos por `random` seja calculada. O **piping** pode ser realizado no UNIX, Linux, Mac OS X e Windows.

A saída de programa pode ser redirecionada para um arquivo utilizando o **símbolo de redirecionamento de saída** (`>`). (O mesmo símbolo é utilizado para UNIX, Linux, Mac OS X e Windows.) Por exemplo, para redirecionar a saída de um programa `random` para um novo arquivo chamado `out`, utilize

```
$ random > out
```



Por fim, a saída do programa pode ser acrescentada ao fim de um arquivo existente utilizando o **símbolo de acréscimo à saída (append output) (>>)**. (O mesmo símbolo é utilizado para UNIX, Linux, Mac OS X e Windows.) Por exemplo, para acrescentar a saída do programa `random` ao arquivo `out` criado na linha de comando anterior, utilize a linha de comando

```
$ random >> out
```

### E.3 Listas de argumentos de comprimento variável<sup>1</sup>

É possível criar funções que recebem um número não especificado de argumentos. Reticências (...) no protótipo de uma função indicam que a função recebe um número variável de argumentos de qualquer tipo. Observe que as reticências devem ser sempre colocadas no fim da lista de parâmetros e deve haver pelo menos um argumento antes das reticências. As macros e definições do **cabeçalho de argumentos variáveis <stdarg.h>** (Figura E.1) fornecem as capacidades necessárias para construir funções com listas de argumentos de comprimento variável.

A Figura E.2 demonstra a função `average`, que recebe um número variável de argumentos. O primeiro argumento de `average` é sempre o número de valores a ter a média calculada, e todos os demais argumentos devem ser do tipo `double`.

A função `average` utiliza todas as definições e macros de cabeçalho `<stdarg.h>`. O objeto `list`, do tipo `va_list`, é utilizado pelas macros `va_start`, `va_arg` e `va_end` para processar a lista de argumentos de comprimento variável da função `average`. A função invoca `va_start` a fim de inicializar o objeto `list` para uso em `va_arg` e `va_end`. A macro aceita dois argumentos — o objeto `list` e o identificador do argumento na extrema direita na lista de argumentos antes das reticências — `count`, nesse caso (`va_start` utiliza `count` aqui para determinar onde a lista de argumentos de comprimento variável inicia).

Em seguida, a função `average` adiciona repetidamente os argumentos na lista de argumentos de comprimento variável ao `total`. O valor a ser adicionado a `total` é recuperado da lista de argumentos invocando a macro `va_arg`. A macro `va_arg` aceita dois argumentos — o objeto `list` e o tipo do valor esperado na lista de argumentos (`double`, nesse caso) — e retorna o valor do argumento. A função `average` invoca a macro `va_end` com o objeto `list` como um argumento antes de retornar. Por fim, a média é calculada e retornada a `main`. Observe que utilizamos somente argumentos `double` para a parte de comprimento variável da lista de argumentos.

Listas de argumentos de comprimento variável promovem as variáveis do tipo `float` ao tipo `double`. Essas listas de argumentos também promovem variáveis integrais que são menores que `int` ao tipo `int` (variáveis do tipo `int`, `unsigned`, `long` e `unsigned long` são deixadas isoladas).



#### Observação de engenharia de software E.1

Listas de argumentos de comprimento variável podem ser utilizadas somente com argumentos do tipo fundamental e do tipo `structs` no estilo C que não contêm recursos específicos do C++ como funções `virtual`, construtores, destrutores, referências, membros de dados `const` e classes `virtual` básicas.



#### Erro comum de programação E.1

Colocar reticências no meio de uma lista de parâmetros de função é um erro de sintaxe. As reticências só podem ser colocadas no fim da lista de parâmetros.

Identificador	Descrição
<code>va_list</code>	Um tipo adequado para armazenar informações necessárias às macros <code>va_start</code> , <code>va_arg</code> e <code>va_end</code> . Para acessar os argumentos em uma lista de argumentos de comprimento variável, um objeto do tipo <code>va_list</code> deve ser declarado.
<code>va_start</code>	Uma macro que é invocada antes dos argumentos de uma lista de argumentos de comprimento variável pode ser acessada. A macro inicializa o objeto declarado com <code>va_list</code> para uso pelas macros <code>va_arg</code> e <code>va_end</code> .
<code>va_arg</code>	Uma macro que se expande para uma expressão do valor e tipo do próximo argumento na lista de argumentos de comprimento variável. Cada invocação de <code>va_arg</code> modifica o objeto declarado com <code>va_list</code> a fim de que o objeto aponte para o próximo argumento na lista.
<code>va_end</code>	Uma macro que realiza faxina de terminação em uma função cuja lista de argumentos de comprimento variável era referenciada pela macro <code>va_start</code> .

**Figura E.1** O tipo e as macros definidas no cabeçalho `<stdarg.h>`.

<sup>1</sup> No C++, os programadores utilizam a sobrecarga de funções para realizar muito do que os programadores C conseguem com as listas de argumentos de comprimento variável.

```

1 // Figura E.2: figE_02.cpp
2 // Utilizando listas de argumentos de comprimento variável.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::ios;
7
8 #include <iomanip>
9 using std::setw;
10 using std::setprecision;
11 using std::setiosflags;
12 using std::fixed;
13
14 #include <cstdarg>
15 using std::va_list;
16
17 double average(int, ...);
18
19 int main()
20 {
21 double double1 = 37.5;
22 double double2 = 22.5;
23 double double3 = 1.7;
24 double double4 = 10.2;
25
26 cout << fixed << setprecision(1) << "double1 = "
27 << double1 << "\ndouble2 = " << double2 << "\ndouble3 = "
28 << double3 << "\ndouble4 = " << double4 << endl
29 << setprecision(3)
30 << "\nThe average of double1 and double2 is "
31 << average(2, double1, double2)
32 << "\nThe average of double1, double2, and double3 is "
33 << average(3, double1, double2, double3)
34 << "\nThe average of double1, double2, double3"
35 << " and double4 is "
36 << average(4, double1, double2, double3, double4)
37 << endl;
38 return 0;
39 } // fim de main
40
41 // calcula a média
42 double average(int count, ...)
43 {
44 double total = 0;
45 va_list list; // para armazenar informações necessárias a va_start
46
47 va_start(list, count);
48
49 // processa lista de argumentos de comprimento variável
50 for (int i = 1; i <= count; i++)
51 total += va_arg(list, double);
52
53 va_end(list); // fim de va_start
54 return total / count;
55 } // fim da função average

```

Figura E.2 Utilizando listas de argumentos de comprimento variável.

(continua)

```
double1 = 37.5
double2 = 22.5
double3 = 1.7
double4 = 10.2

The average of double1 and double2 is 30.000
The average of double1, double2, and double3 is 20.567
The average of double1, double2, double3 and double4 is 17.975
```

**Figura E.2** Utilizando listas de argumentos de comprimento variável.

(continuação)

## E.4 Utilizando argumentos de linha de comando

Em muitos sistemas — Windows, UNIX, Linux e Mac OS X em particular — é possível passar argumentos para `main` a partir de uma linha de comando incluindo os parâmetros `int argc` e `char *argv[]` na lista de parâmetros de `main`. O parâmetro `argc` recebe o número de argumentos de linha de comando. O parâmetro `argv` é um array de `char *`s que apontam para strings em que os argumentos de linha de comando reais são armazenados. Usos comuns de argumentos de linha de comando incluem imprimir os argumentos, passar opções para um programa e passar nomes de arquivo para um programa.

A Figura E.3 copia um arquivo para outro arquivo um caractere por vez. O arquivo executável para o programa é chamado `copyFile` (isto é, o nome executável para o arquivo). Uma linha de comando típica para o programa `copyFile` em um sistema de UNIX é

```
$ copyFile input output
```

Essa linha de comando indica que o arquivo `input` deve ser copiado para o arquivo `output`. Quando o programa executar, se `argc` não for 3 (`copyFile` conta como um dos argumentos), o programa imprimirá uma mensagem de erro (linha 16). Caso contrário, o array `argv` conterá as strings `"copyFile"`, `"input"` e `"output"`. O segundo e o terceiro argumentos na linha de comando são utilizados como nomes de arquivo pelo programa. Os arquivos são abertos criando o objeto `ifstream inFile` e o objeto `ofstream outFile` (linhas 19 e 28). Se ambos os arquivos forem abertos com sucesso, os caracteres serão lidos a partir do arquivo `input` com a função-membro `get` e gravados no arquivo `output` com a função-membro `put` até que o indicador de fim de arquivo do arquivo `input` seja configurado (linhas 40–44). Então o programa termina. O resultado é uma cópia exata do arquivo `input`. Observe que nem todos os sistemas de computador suportam argumentos de linha de comando tão facilmente quanto os sistemas UNIX, Linux, Mac OS X e Windows. Alguns VMS, e sistemas Macintosh mais antigos, por exemplo, requerem configurações especiais para processamento de argumentos de linha de comando. Consulte os manuais de sistema para obter informações adicionais sobre argumentos de linha de comando.

## E.5 Notas sobre a compilação de programas de arquivo de múltiplas fontes

Como declarado anteriormente no texto, é normal construir programas que consistem em múltiplos arquivos-fonte (ver o Capítulo 9, “Classes: um exame mais profundo, parte 1”). Há várias considerações ao criar programas em múltiplos arquivos. Por exemplo, a definição de uma função deve estar inteiramente contida em um arquivo — ela não pode distribuir dois ou mais arquivos.

No Capítulo 6, introduzimos os conceitos de escopo e classe de armazenamento. Aprendemos que as variáveis declaradas fora de qualquer definição de função são da classe de armazenamento `static` por padrão e são referidas como variáveis globais. As variáveis globais são acessíveis a qualquer função definida no mesmo arquivo depois que a variável é declarada. As variáveis globais também são funções acessíveis em outros arquivos; entretanto, as variáveis globais devem ser declaradas em cada arquivo em que são utilizadas. Por exemplo, se definimos a variável global `flag` do tipo inteiro em um arquivo e a referenciarmos em um segundo arquivo, este deve conter a declaração

```
extern int flag;
```

antes do uso da variável nesse arquivo. Na declaração anterior, o especificador de classe de armazenamento `extern` indica ao compilador que a variável `flag` é definida mais tarde no mesmo arquivo ou em um arquivo diferente. O compilador informa o linker de que as referências não resolvidas à variável `flag` aparecem no arquivo. (Como o compilador não sabe onde o `flag` está definido, ele deixa o linker tentar localizá-lo.) Se o linker não puder localizar uma definição de `flag`, um erro de linker é informado. Se uma definição global adequada é localizada, o linker resolve as referências indicando a localização do `flag`.



### Dica de desempenho E.1

*As variáveis globais aumentam o desempenho porque podem ser acessadas diretamente por qualquer função — o overhead de passar dados para funções é eliminado.*

```

1 // Figura E.3: figE_03.cpp
2 // Utilizando argumentos de linha de comando
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::ios;
7
8 #include <fstream>
9 using std::ifstream;
10 using std::ofstream;
11
12 int main(int argc, char *argv[])
13 {
14 // verifica número de argumentos de linha de comando
15 if (argc != 3)
16 cout << "Usage: copyFile infile_name outfile_name" << endl;
17 else
18 {
19 ifstream inFile(argv[1], ios::in);
20
21 // o arquivo de entrada não pôde ser aberto
22 if (!inFile)
23 {
24 cout << argv[1] << " could not be opened" << endl;
25 return -1;
26 } // fim do if
27
28 ofstream outFile(argv[2], ios::out);
29
30 // o arquivo de saída não pôde ser aberto
31 if (!outFile)
32 {
33 cout << argv[2] << " could not be opened" << endl;
34 inFile.close();
35 return -2;
36 } // fim do if
37
38 char c = inFile.get(); // lê o primeiro caractere
39
40 while (inFile)
41 {
42 outFile.put(c); // o caractere de saída
43 c = inFile.get(); // lê o próximo caractere
44 } // fim do while
45 } // fim do else
46
47 return 0;
48 } // fim do main

```

**Figura E.3** Utilizando argumentos de linha de comando.



## Observação de engenharia de software E.2

*As variáveis globais devem ser evitadas a menos que o desempenho do aplicativo seja crítico ou a variável represente um recurso global compartilhado como `c++`, porque eles violam o princípio do menor privilégio e dificultam a manutenção do software.*

Assim como as declarações extern podem ser utilizadas a fim de declarar variáveis globais para outros arquivos de programa, os protótipos de função podem ser utilizados para declarar as funções em outros arquivos de programa. (O especificador extern não é requere-

rido em um protótipo de função.) Isso é realizado incluindo o protótipo de função em cada arquivo em que a função é invocada e, então, compilando cada arquivo-fonte e vinculando os arquivos de código-objeto resultantes. Os protótipos de função indicam ao compilador que a função especificada é definida mais tarde no mesmo arquivo ou em um arquivo diferente. O compilador não tenta resolver referências para tal função — essa tarefa é deixada para o linker. Se o linker não puder localizar uma definição de função, um erro é gerado.

Como um exemplo do uso de protótipos de função para estender o escopo de uma função, considere qualquer programa contendo uma diretiva de pré-processador. Essa diretiva inclui em um arquivo os protótipos de função para funções como `strcmp` e `strcat`. Outras funções no arquivo podem utilizar `strcmp` e `strcat` para realizar suas tarefas. As funções `strcmp` e `strcat` já estão definidas separadamente. Não precisamos saber onde elas são definidas. Estamos simplesmente reutilizando o código em nossos programas. O linker resolve nossas referências para essas funções. Esse processo permite utilizar as funções da biblioteca-padrão.



### Observação de engenharia de software E.3

*Criar programas em múltiplos arquivos-fonte facilita a reusabilidade e a boa engenharia de softwares. As funções podem ser comuns a muitos aplicativos. Nesses casos, essas funções devem ser armazenadas nos próprios arquivos-fonte, e cada arquivo-fonte deve ter um arquivo de cabeçalho correspondente contendo protótipos de função. Isso permite aos programadores de diferentes aplicativos reutilizar o mesmo código incluindo o arquivo de cabeçalho adequado e compilando o aplicativo com o arquivo-fonte correspondente.*



### Dica de portabilidade E.1

*Alguns sistemas não suportam nomes de variável global ou nomes de função que tenham mais de seis caracteres. Isso deve ser considerado ao escrever programas que serão portados para múltiplas plataformas.*

É possível restringir o escopo de uma variável global ou função ao arquivo em que ele é definido. O especificador de classe de armazenamento `static`, quando aplicado a uma variável de escopo de arquivo ou a uma função, impede que ele seja utilizado por qualquer função que não seja definida no mesmo arquivo. Isso é referido como **linkagem interna**. As variáveis globais (exceto as que são `const`) e as funções que não são precedidas por `static` em suas definições têm **linkagem externa** — elas podem ser acessadas em outros arquivos se esses arquivos contiverem declarações e/ou protótipos de função adequados.

A declaração da variável global

```
static double pi = 3.14159;
```

cria a variável `pi` do tipo `double`, inicializa essa variável como 3.14159 e indica que a variável `pi` é conhecida somente por funções no arquivo em que ela é definida.

O especificador `static` é comumente utilizado com funções utilitárias que são chamadas apenas por funções em um arquivo particular. Se uma função não for requerida fora de um arquivo particular, o princípio do menor privilégio deve ser imposto utilizando `static`. Se uma função é definida antes de ser utilizada em um arquivo, `static` deve ser aplicada à definição de função. Caso contrário, `static` deve ser aplicada ao protótipo de função.

Ao construir programas grandes a partir de múltiplos arquivos-fonte, a compilação do programa torna-se tediosa se pequenas alterações em um arquivo implicarem a recompilação de todo o programa. Muitos sistemas fornecem utilitários especiais que recompilam apenas arquivos-fonte dependentes do arquivo de programa modificado. Em sistemas UNIX, o utilitário é chamado **make**. O utilitário **make** lê um arquivo chamado **Makefile** que contém instruções para compilar e vincular o programa. Sistemas como Borland C++ e Microsoft Visual C++ para PCs fornecem os utilitários **make** e 'projetos'. Para obter informações adicionais sobre os utilitários **make**, veja o manual do seu sistema particular.

## E.6 Terminação de programa com `exit` e `atexit`

A biblioteca geral de utilitários (`<stdlib.h>`) fornece métodos para terminar a execução de programas diferentes daqueles de um retorno convencional de função `main`. A função **exit** força um programa a terminar como se ele executasse normalmente. A função é frequentemente utilizada para terminar um programa quando um erro é detectado na entrada ou quando não é possível abrir um arquivo a ser processado pelo programa.

A função **atexit** registra uma função no programa a ser chamada quando o programa termina alcançando o fim de `main` ou quando a função `exit` é invocada. A função `atexit` aceita um ponteiro para uma função (isto é, o nome de função) como um argumento. As funções chamadas na terminação do programa não podem ter argumentos e não podem retornar um valor.

A função `exit` aceita um argumento. O argumento é normalmente a constante simbólica **EXIT\_SUCCESS** ou **EXIT\_FAILURE**. Se `exit` for chamada com **EXIT\_SUCCESS**, o valor definido pela implementação para terminação bem-sucedida é retornado ao ambiente chamador. Se `exit` for chamada com **EXIT\_FAILURE**, o valor definido pela implementação para a terminação malsucedida é retornado. Quando a função `exit` é invocada, quaisquer funções anteriormente registradas com a função `atexit` são invocadas na ordem inversa de seu registro, todos os fluxos associados com o programa são esvaziados e fechados, e o controle retorna ao ambiente `host`. A Figura E.4 testa as funções `exit` e `atexit`. O programa pede para o usuário determinar se o programa deve ou não ser terminado com `exit` ou alcançando o fim de `main`. Observe que a função `print` é executada ao término do programa em cada caso.

```

1 // Figura E.4: figE_04.cpp
2 // Utilizando as funções exit e atexit
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::cin;
7
8 #include <cstdlib>
9 using std::atexit;
10 using std::exit;
11
12 void print();
13
14 int main()
15 {
16 atexit(print); // registra a função print
17
18 cout << "Enter 1 to terminate program with function exit"
19 << "\nEnter 2 to terminate program normally\n";
20
21 int answer;
22 cin >> answer;
23
24 // encerra se a resposta for 1
25 if (answer == 1)
26 {
27 cout << "\nTerminating program with function exit\n";
28 exit(EXIT_SUCCESS);
29 } // fim do if
30
31 cout << "\nTerminating program by reaching the end of main"
32 << endl;
33
34 return 0;
35 } // fim de main
36
37 // exibe a mensagem antes da terminação
38 void print()
39 {
40 cout << "Executing function print at program termination\n"
41 << "Program terminated" << endl;
42 } // fim da função print

```

```

Enter 1 to terminate program with function exit
Enter 2 to terminate program normally
2

```

```

Terminating program by reaching the end of main
Executing function print at program termination
Program terminated

```

```

Enter 1 to terminate program with function exit
Enter 2 to terminate program normally
1

```

**Figura E.4** Utilizando as funções exit e atexit.

(continua)

```
Terminating program with function exit
Executing function print at program termination
Program terminated
```

**Figura E.4** Utilizando as funções `exit` e `atexit`.

(continuação)

## E.7 O qualificador de tipo `volatile`

O qualificador de tipo `volatile` é aplicado a uma definição de uma variável que pode ser alterada fora do programa (isto é, a variável não está completamente sob o controle do programa). Portanto, o compilador não pode realizar otimizações (como acelerar a execução de programa ou reduzir o consumo de memória) que dependam de ‘saber que o comportamento de uma variável é influenciado somente por atividades de programa que o compilador pode observar’.

## E.8 Sufixos para constantes de inteiro e de ponto flutuante

O C++ fornece sufixos do tipo inteiro e de ponto flutuante para especificar os tipos de constantes inteiras e de ponto flutuante. Os sufixos de inteiro são: `u` ou `U` para um inteiro `unsigned`, `l` ou `L` para um inteiro `long` e `ul` ou `UL` para um inteiro `unsigned long`. As seguintes constantes são do tipo `unsigned`, `long` e `unsigned long`, respectivamente:

```
174u
8358L
28373ul
```

Se uma constante do tipo inteiro não estiver sufixada, seu tipo é `int`; se a constante não puder ser armazenada em um `int`, ela é armazenada em um `long`.

Os sufixos de ponto flutuante são `f` ou `F` para um `float` e `l` ou `L` para um `long double`. As seguintes constantes são do tipo `long double` e `float`, respectivamente:

```
3.14159L
1.28f
```

Uma constante de ponto flutuante que não está sufixada é de tipo `double`. Um constante com um sufixo inadequado resulta em um aviso ou erro de compilador.

## E.9 Tratamento de sinal

Um evento inesperado, ou **sinal**, pode terminar um programa prematuramente. Alguns eventos inesperados incluem **interrupções** (pressionar `Ctrl+C` em um sistema UNIX, Linux, Mac OS X ou Windows), **instruções ilegais**, **violações de segmentação**, **pedidos de terminação provenientes do sistema operacional** e **exceções de ponto flutuante** (divisão por zero ou multiplicação de grandes valores de ponto flutuante). A **biblioteca de tratamento de sinal** fornece a função `signal` para **interceptar eventos inesperados**. A função `signal` recebe dois argumentos — um número de sinal inteiro e um ponteiro para a função de tratamento de sinal. Os sinais podem ser gerados pela função `raise`, que aceita um número de sinal do tipo inteiro como um argumento. A Figura E.5 resume os sinais-padrão definidos em arquivo de cabeçalho `<csignal>`. O próximo exemplo demonstra as funções `signal` e `raise`.

A Figura E.6 intercepta um sinal interativo (`SIGINT`) com a função `signal`. O programa chama `signal` com `SIGINT` e um ponteiro para a função `signalHandler`. (Lembre-se de que o nome de uma função é um ponteiro para a função.) Agora, quando ocorre um sinal do tipo `SIGINT`, a função `signalHandler` é chamada, uma mensagem é impressa e o usuário recebe a opção para continuar a execução

Sinal	Explicação
<code>SIGABRT</code>	Terminação anormal do programa (como uma chamada para <code>abort</code> ).
<code>SIGFPE</code>	Uma operação aritmética errônea, como uma divisão por zero ou uma operação que resulte em estouro.
<code>SIGILL</code>	Deteção de uma instrução ilegal.
<code>SIGINT</code>	Recebimento de um sinal de alerta interativo.
<code>SIGSEGV</code>	Um acesso inválido para armazenamento.
<code>SIGTERM</code>	Uma solicitação de terminação enviada para o programa.

**Figura E.5** Sinais definidos no cabeçalho `<csignal>`.



normal do programa. Se o usuário quiser continuar a execução, o handler de sinal é reinicializado chamando `signal` novamente (alguns sistemas requerem que o handler de sinal seja reinicializado) e o controle retorna ao ponto no programa em que o sinal foi detectado. Nesse programa, a função `raise` é utilizada para simular um sinal interativo. Um número aleatório entre 1 e 50 é escolhido. Se o número é 25, então `raise` é chamado para gerar o sinal. Normalmente, sinais interativos são iniciados fora do programa. Por exemplo, pressionar `Ctrl+C` durante a execução de programa em um sistema UNIX, Linux, Mac OS X ou Windows gera um sinal interativo que termina a execução do programa. O tratamento de sinal pode ser utilizado para interceptar o sinal interativo e impede a terminação do programa.

```

1 // Figura E.6: figE_06.cpp
2 // Utilizando o tratamento de sinal
3 #include <iostream>
4 using std::cout;
5 using std::cin;
6 using std::endl;
7
8 #include <iomanip>
9 using std::setw;
10
11 #include <csignal>
12 using std::raise;
13 using std::signal;
14
15 #include <cstdlib>
16 using std::exit;
17 using std::rand;
18 using std::srand;
19
20 #include <ctime>
21 using std::time;
22
23 void signalHandler(int);
24
25 int main()
26 {
27 signal(SIGINT, signalHandler);
28 srand(time(0));
29
30 // cria e gera saída de números aleatórios
31 for (int i = 1; i <= 100; i++)
32 {
33 int x = 1 + rand() % 50;
34
35 if (x == 25)
36 raise(SIGINT); // levanta SIGINT quando x é 25
37
38 cout << setw(4) << i;
39
40 if (i % 10 == 0)
41 cout << endl; // gera saída de endl quando i é um múltiplo de 10
42 } // fim do for
43
44 return 0;
45 } // fim de main
46
47 // trata o sinal

```

**Figura E.6** Utilizando o tratamento de sinal.

(continua)

```

48 void signalHandler(int signalValue)
49 {
50 cout << "\nInterrupt signal (" << signalValue
51 << ") received.\n"
52 << "Do you wish to continue (1 = yes or 2 = no)? ";
53
54 int response;
55
56 cin >> response;
57
58 // verifica respostas inválidas
59 while (response != 1 && response != 2)
60 {
61 cout << "(1 = yes or 2 = no)? ";
62 cin >> response;
63 } // fim do while
64
65 // determina se é hora de sair
66 if (response != 1)
67 exit(EXIT_SUCCESS);
68
69 // chama signal e lhe passa SIGINT e o endereço de signalHandler
70 signal(SIGINT, signalHandler);
71 } // fim da função signalHandler

```

```

1 2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50
51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 68 69 70
71 72 73 74 75 76 77 78 79 80
81 82 83 84 85 86 87 88 89 90
91 92 93 94 95 96 97 98 99

```

```

Interrupt signal (2) received.
Do you wish to continue (1 = yes or 2 = no)? 1
100

```

```

1 2 3 4
Interrupt signal (2) received.
Do you wish to continue (1 = yes or 2 = no)? 2

```

**Figura E.6** Utilizando o tratamento de sinal.

(continuação)

## E.10 Alocação dinâmica de memória com calloc e realloc

No Capítulo 10, discutimos a alocação dinâmica de memória no estilo C++ com `new` e `delete`. Os programadores em C++ devem utilizar `new` e `delete`, em vez das funções do C `malloc` e `free` (cabeçalho `<cstdlib>`). Entretanto, a maioria dos programadores em C++ se verá lendo uma grande quantidade de código C legado, portanto incluímos essa discussão adicional sobre alocação dinâmica de memória no estilo C.

A biblioteca geral de utilitários (`<cstdlib>`) fornece outras duas funções para alocação dinâmica de memória — `calloc` e `realloc`. Essas funções podem ser utilizadas para criar e modificar **arrays dinâmicos**. Como mostrado no Capítulo 8, “Ponteiros e strings baseadas em ponteiro”, um ponteiro para um array pode ser subscrito como um array. Portanto, um ponteiro para uma parte contígua da memória criada por `calloc` pode ser manipulado como um array. A função `calloc` aloca dinamicamente a memória para um array e inicializa a memória como zeros. O protótipo para `calloc` é

```
void *calloc(size_t nmemb, size_t size);
```

Essa função recebe dois argumentos — o número de elementos (`nmemb`) e o tamanho de cada elemento (`size`) — e inicializa os elementos do array como zero. A função retorna um ponteiro para a memória alocada ou para um ponteiro nulo (0) se a memória não for alocada.

A função `realloc` muda o tamanho de um objeto alocado por uma chamada anterior a `malloc`, `calloc` ou `realloc`. O conteúdo do objeto original não é modificado, visto que a memória alocada é maior que a quantidade alocada anteriormente. Caso contrário, o conteúdo permanece inalterado até o tamanho do novo objeto. O protótipo para `realloc` é

```
void *realloc(void *ptr, size_t size);
```

A função `realloc` aceita dois argumentos — um ponteiro para o objeto original (`ptr`) e o novo tamanho do objeto (`size`). Se `ptr` for 0, `realloc` funciona de modo idêntico a `malloc`. Se `size` for 0 e `ptr` não for 0, a memória para o objeto é liberada. Caso contrário, se `ptr` não for 0 e o tamanho for maior que zero, `realloc` tenta alocar um novo bloco de memória. Se o novo espaço não puder ser alocado, o objeto apontado por `ptr` permanecerá inalterado. A função `realloc` retorna um ponteiro para a memória realocada ou um ponteiro nulo.



## Erro comum de programação E.2

*Utilizar o operador delete em um ponteiro criado por malloc, calloc e realloc. Utilizar realloc ou free em um ponteiro criado com o uso do operador new.*

## E.11 O desvio incondicional: goto

Por todo o texto salientamos a importância do uso de técnicas de programação estruturada para construir softwares confiáveis que são fáceis de depurar, manter e modificar. Em alguns casos, o desempenho é mais importante que a estrita obediência a técnicas de programação estruturada. Nesses casos, algumas técnicas de programação não estruturadas podem ser utilizadas. Por exemplo, podemos utilizar `break` para terminar a execução de uma estrutura de repetição antes de a condição de continuação do loop tornar-se falsa. Isso poupa repetições desnecessárias do loop se a tarefa estiver completada antes da terminação do loop.

Outro caso de programação não estruturada é a **instrução goto** — um desvio incondicional. O resultado da instrução `goto` é uma mudança no fluxo de controle do programa para a primeira instrução depois do **rótulo** especificado na instrução `goto`. Um rótulo é um identificador seguido por dois-pontos. Um rótulo deve aparecer na mesma função que a instrução `goto` que ele referencia. A Figura E.7 utiliza as instruções `goto` para fazer loop 10 vezes e imprimir o valor de contador toda vez. Depois de inicializar `count` como 1, o programa testa `count` para determinar se ele é maior que 10. (O rótulo `start` é pulado, porque os rótulos não executam nenhuma ação.) Se for, o controle é transferido de `goto` para a primeira instrução depois do rótulo `end`. Caso contrário, `count` é impresso e incrementado, e o controle é transferido da instrução `goto` para a primeira instrução depois do rótulo `start`.

Nos capítulos 4–5, declaramos que apenas três estruturas de controle são necessárias para escrever qualquer programa — a sequência, a seleção e a repetição. Quando as regras de programação estruturada são seguidas, é possível criar estruturas de controle profundamente aninhadas a partir das quais é difícil de escapar eficientemente. Alguns programadores utilizam instruções `goto` nessas situações como uma saída rápida de uma estrutura profundamente aninhada. Isso elimina a necessidade de testar múltiplas condições para escapar de uma estrutura de controle.



## Dica de desempenho E.2

*A instrução goto pode ser utilizada para sair de estruturas de controle profundamente aninhadas de maneira eficiente, mas pode dificultar a leitura e a manutenção do código.*



## Dica de prevenção de erro E.1

*A instrução goto só deve ser utilizada em aplicativos orientados ao desempenho. A instrução goto não tem estrutura e pode resultar em programas que são mais difíceis de depurar, manter e modificar.*

## E.12 Uniões

Uma **união** (definida com a palavra-chave `union`) é uma região de memória que, com o passar do tempo, pode conter objetos de uma variedade de tipos. Entretanto, a qualquer momento, uma `union` pode conter um máximo de um objeto, porque os membros de uma `union` compartilham o mesmo espaço de armazenamento. É responsabilidade do programador assegurar que os dados em uma `union` sejam referenciados com um nome de membro do tipo de dados adequado.



## Erro comum de programação E.3

*O resultado de referenciar um membro union diferente do último armazenado é indefinido. Ele trata os dados armazenados como um tipo diferente.*

```

1 // Figura E.7: figE_07.cpp
2 // Utilizando goto.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include <iomanip>
8 using std::left;
9 using std::setw;
10
11 int main()
12 {
13 int count = 1;
14
15 start: // rótulo
16 // goto termina quando a contagem excede 10
17 if (count > 10)
18 goto end;
19
20 cout << setw(2) << left << count;
21 ++count;
22
23 // goto inicia na linha 17
24 goto start;
25
26 end: // rótulo
27 cout << endl;
28
29 return 0;
30 } // fim do main

```

1 2 3 4 5 6 7 8 9 10

**Figura E.7** Utilizando goto.



### Dica de portabilidade E.2

*Se os dados são armazenados em uma union como um tipo e referenciados como outro tipo, os resultados são dependentes da implementação.*

Em momentos diferentes durante a execução de um programa, alguns objetos talvez não sejam relevantes, embora outro objeto seja — portanto, uma union compartilha o espaço em vez de desperdiçar armazenamento com objetos que não estão sendo utilizados. O número de bytes utilizado para armazenar um union será suficiente pelo menos para armazenar o membro maior.



### Dica de desempenho E.3

*Utilizar unions economiza o armazenamento.*



### Dica de portabilidade E.3

*A quantidade de armazenamento necessária para armazenar uma union é dependente da implementação.*

Uma union é declarada no mesmo formato que uma struct ou uma class. Por exemplo,

```

union Number
{
 int x;
 double y;
};

```

indica que `Number` é um tipo `union` com membros `int x` e `double y`. A definição `union` deve preceder todas as funções em que serão utilizadas.



### Observação de engenharia de software E.4

*Como um `struct` ou uma declaração `class`, uma declaração `union` simplesmente cria um novo tipo. Colocar uma declaração `union struct` fora de qualquer função não cria uma variável global.*

As únicas operações predefinidas válidas que podem ser realizadas em uma `union` são atribuir uma `union` a outra `union` do mesmo tipo, aceitar o endereço (&) de uma `union` e acessar os membros `union` que utilizam o operador de membro de estrutura (.) e o operador de ponteiro de estrutura (->). As `unions` não podem ser comparadas.



### Erro comum de programação E.4

*Comparar `unions` é um erro de compilação, porque o compilador não sabe qual membro de cada uma está ativo e, portanto, qual membro de uma comparar com qual membro da outra.*

Uma `union` é semelhante a uma classe no sentido de que ela pode ter um construtor para inicializar qualquer um de seus membros. Uma `union` que não tem nenhum construtor pode ser inicializada com outra `union` do mesmo tipo, com uma expressão do tipo do primeiro membro da `union` ou com um inicializador (entre chaves) do tipo do primeiro membro da `union`. As `unions` podem ter outras funções-membro, como destrutores, mas as funções-membro de uma `union` não podem ser declaradas `virtual`. Os membros de uma `union` são `public` por padrão.



### Erro comum de programação E.5

*Inicializar uma `union` em uma declaração com um valor ou uma expressão cujo tipo é diferente do tipo do primeiro membro da `union` é um erro de compilação.*

Uma `union` não pode ser utilizada como uma classe básica em herança (isto é, as classes não podem ser derivadas de `unions`). As `unions` podem ter objetos como membros somente se esses objetos não tiverem um construtor, um destrutor ou um operador de atribuição sobrecarregado. Nenhum dos membros de dados de uma `union` pode ser declarado `static`.

A Figura E.8 utiliza a variável `value` do tipo `union Number` para exibir o valor armazenado na `union` como um `int` e como um `double`. A saída de programa é dependente de implementação. A saída do programa mostra que a representação interna de um valor `double` pode ser bem diferente da representação de um `int`.

Uma `union` **anônima** é uma `union` sem nome do tipo que não tenta definir objetos ou ponteiros antes de seu ponto-e-vírgula de terminação. Tal `union` não cria um tipo, mas cria objeto não identificado (não nomeado). Os membros de uma `union` anônima podem ser acessados diretamente no escopo em que a `union` anônima é declarada assim como qualquer outra variável local — não há necessidade de utilizar os operadores de ponto (.) ou seta (->).

As `unions` anônimas têm algumas restrições. As `unions` anônimas podem conter apenas membros de dados. Todos os membros de uma `union` anônima devem ser `public`. E uma `union` anônima declarada globalmente (isto é, no escopo de arquivo) deve ser declarada explicitamente como `static`. A Figura E.9 ilustra o uso de uma `union` anônima.

```

1 // Figura E.8: figE_08.cpp
2 // Um exemplo de uma união.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 // define a union Number
8 union Number
9 {
10 int integer1;
11 double double1;
12 }; // fim da union Number
13
14 int main()
15 {
16 Number value; // variável de união

```

**Figura E.8** Imprimindo o valor de uma união em ambos os tipos de dados de membro.

(continua)

```

17
18 value.integer1 = 100; // atribui 100 ao membro integer1
19
20 cout << "Put a value in the integer member\n"
21 << "and print both members.\nint: "
22 << value.integer1 << "\ndouble: " << value.double1
23 << endl;
24
25 value.double1 = 100.0; // atribui 100.0 ao membro double1
26
27 cout << "Put a value in the floating member\n"
28 << "and print both members.\nint: "
29 << value.integer1 << "\ndouble: " << value.double1
30 << endl;
31
32 return 0;
33 } // fim de main

```

```

Put a value in the integer member
and print both members.
int: 100
double: -9.25596e+061
Put a value in the floating member
and print both members.
int: 0
double: 100

```

**Figura E.8** Imprimindo o valor de uma união em ambos os tipos de dados de membro.

(continuação)

```

1 // Figura E.9: figE_09.cpp
2 // Utilizando uma union anônima.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 int main()
8 {
9 // declara uma union anônima
10 // os membros integer1, double1 e charPtr compartilham o mesmo espaço
11 union
12 {
13 int integer1;
14 double double1;
15 char *charPtr;
16 }; // fim da union anônima
17
18 // declara as variáveis locais
19 int integer2 = 1;
20 double double2 = 3.3;
21 char *char2Ptr = "Anonymous";
22
23 // atribui o valor a cada membro union
24 // sucessivamente e imprime cada um
25 cout << integer2 << ' ';

```

**Figura E.9** Utilizando uma union anônima.

(continua)

```

26 integer1 = 2;
27 cout << integer1 << endl;
28
29 cout << double2 << ' ';
30 double1 = 4.4;
31 cout << double1 << endl;
32
33 cout << char2Ptr << ' ';
34 charPtr = "union";
35 cout << charPtr << endl;
36
37 return 0;
38 } // fim do main

```

```

1 2
3.3 4.4
Anonymous union

```

Figura E.9 Utilizando uma union anônima.

(continuação)

### E.13 Especificações de linkagem

É possível, a partir de um programa C++, chamar funções escritas e compiladas com um compilador C. Como declarado na Seção 6.17, o C++ codifica especialmente nomes de função para linkagem fortemente tipada. O C, entretanto, não codifica seus nomes de função. Portanto, uma função compilada em C não será reconhecida quando uma tentativa de linkar o código C com código C++ for feita, porque o código C++ espera um nome de função especialmente codificado. O C++ permite ao programador fornecer **especificações de linkagem** para informar ao compilador que uma função foi compilada em um compilador C e impedir que o nome da função seja codificado pelo compilador C++. As especificações de linkagem são úteis quando grandes bibliotecas de funções especializadas tiverem sido desenvolvidas, e o usuário não tiver acesso ao código-fonte para recompilação no C++ ou não tiver tempo de converter as funções de biblioteca C para C++.

Para informar ao compilador que uma ou várias das funções foram compiladas em C, escreva os protótipos de função como mostrado a seguir:

```

extern "C" protótipo de função // uma única função

extern "C" // múltiplas funções
{
 protótipos de função
}

```

Essas declarações informam ao compilador que as funções especificadas não foram compiladas em C++, então a codificação de nome não deve ser realizada nas funções listadas na especificação de linkagem. Essas funções podem então ser vinculadas adequadamente com o programa. Os ambientes C++ normalmente incluem as bibliotecas C padrão e não requerem que o programador utilize especificações de linkagem para essas funções.

### E.14 Síntese

Este apêndice introduziu vários tópicos sobre o código C legado. Discutimos o redirecionamento da entrada pelo teclado de modo a fazê-la vir de um arquivo e o redirecionamento de saída na tela para um arquivo. Introduzimos também listas de argumentos de comprimento variável, argumentos de linha de comando e processamento de eventos inesperados. Você também aprendeu a alocar e redimensionar memória dinamicamente. No próximo apêndice, você aprenderá a utilizar o pré-processador para incluir outros arquivos, definir constantes simbólicas e macros.



## Resumo

- Em muitos sistemas — sistemas UNIX, Linux, Mac OS X ou Windows em particular — é possível redirecionar a entrada para um programa e a saída a partir de um programa. A entrada é redirecionada a partir das linhas de comando UNIX, Linux, Mac OS X ou Windows utilizando o símbolo de entrada de redirecionamento (<) ou um *pipe* (|). A saída é redirecionada a partir das linhas de comando UNIX, Linux, Mac OS X ou Windows utilizando o símbolo de redirecionamento de saída (>) ou símbolo de acréscimo à saída (>>). O símbolo de redirecionamento da saída simplesmente armazena a saída do programa em um arquivo, e o símbolo de acréscimo da saída adiciona a saída ao fim de um arquivo.
- As macros e as definições do cabeçalho de argumentos variáveis <stdarg.h> fornecem as capacidades necessárias para construir funções com listas de argumentos de comprimento variável.
- Reticências (...) em um protótipo de função indicam que a função recebe um número variável de argumentos.
- O tipo `va_list` é adequado para armazenar informações necessárias às macros `va_start`, `va_arg` e `va_end`. Para acessar os argumentos em uma lista de argumentos de comprimento variável, um objeto do tipo `va_list` deve ser declarado.
- A macro `va_start` é invocada antes que os argumentos de uma lista de argumentos de comprimento variável possam ser acessados. A macro inicializa o objeto declarado com `va_list` para que seja utilizado pelas macros `va_arg` e `va_end`.
- A macro `va_arg` se expande para uma expressão do valor e tipo do próximo argumento na lista de argumentos de comprimento variável. Cada invocação de `va_arg` modifica o objeto `va_list` a fim de que o objeto aponte para o próximo argumento na lista.
- A macro `va_end` facilita um retorno normal de uma função cuja lista de argumentos variável era referenciada pela macro `va_start`.
- Em muitos sistemas — UNIX, Linux, Mac OS X e Windows em particular — é possível passar argumentos de linha de comando para `main` incluindo na lista de parâmetros de `main` os parâmetros `int argc` e `char *argv[]`. O parâmetro `argc` é o número de argumentos de linha de comando. O parâmetro `argv` é um array de `char *`s contendo o argumento de linha de comando.
- A definição de uma função deve estar inteiramente contida em um arquivo — não pode se estender por dois ou mais arquivos.
- As variáveis globais devem ser declaradas em cada arquivo em que são utilizadas.
- Os protótipos de função podem ser utilizados para declarar funções em outros arquivos de programa. (O especificador `extern` não é requerido em um protótipo de função.) Isso é realizado incluindo o protótipo de função em cada arquivo em que a função é invocada e, então, compilando os arquivos juntos.
- O especificador de classe de armazenamento `static`, quando aplicado a uma variável de escopo de arquivo ou a uma função, impede que ele seja utilizado por qualquer função que não seja definida no mesmo arquivo. Isso é referido como linkagem interna. As variáveis globais e funções que não são precedidas por `static` em suas definições têm linkagem externa — elas podem ser acessadas em outros arquivos se esses arquivos contiverem declarações e/ou protótipos de função adequados.
- O especificador `static` é comumente utilizado com funções utilitárias que são chamadas apenas por funções em um arquivo particular. Se uma função não for requerida fora de um arquivo particular, o princípio do menor privilégio deve ser imposto utilizando `static`.
- Ao construir programas grandes a partir de múltiplos arquivos-fonte, a compilação do programa torna-se tediosa se pequenas alterações em um arquivo implicarem a recompilação de todo o programa. Muitos sistemas fornecem utilitários especiais que recompilam somente o arquivo de programa modificado. Em sistemas UNIX, o utilitário é chamado `make`. O utilitário `make` lê um arquivo chamado `Makefile`, que contém instruções para compilar e linkar o programa.
- A função `exit` força um programa a terminar como se ele tivesse executado normalmente.
- A função `atexit` registra uma função em um programa a ser chamada na terminação normal do programa — isto é, quando o programa terminar alcançando o fim de `main` ou quando `exit` for invocada.
- A função `atexit` aceita um ponteiro para uma função (por exemplo, um nome de função) como um argumento. As funções chamadas na terminação do programa não podem ter argumentos e não pode retornar um valor.
- A função `exit` aceita um argumento — normalmente a constante simbólica `EXIT_SUCCESS` ou a constante simbólica `EXIT_FAILURE`. Se `exit` for chamada com `EXIT_SUCCESS`, o valor definido pela implementação para a terminação bem-sucedida é retornado ao ambiente chamador. Se `exit` for chamada com `EXIT_FAILURE`, o valor definido pela implementação para a terminação malsucedida é retornado.
- Quando a função `exit` é invocada, todas as funções registradas com `atexit` são invocadas na ordem inversa de seu registro, todos os fluxos associados com o programa são esvaziados e fechados e o controle retorna ao ambiente de `host`.
- O qualificador `volatile` é utilizado para impedir otimizações de uma variável, porque ela pode ser modificada de fora do escopo do programa.
- O C++ fornece sufixos do tipo inteiro e de ponto flutuante para especificar os tipos de constantes inteiras e de ponto flutuante. Os sufixos de inteiro são `u` ou `U` para um inteiro `unsigned`, `l` ou `L` para um inteiro `long` e `ul` ou `UL` para um inteiro `unsigned long`. Se uma constante do tipo inteiro não estiver sufixada, seu tipo é determinado pelo primeiro tipo capaz de armazenar um valor desse tamanho (primeiro `int` e, depois, `long int`). Os sufixos de ponto flutuante são `f` ou `F` para um `float` e `l` ou `L` para um `long double`. Uma constante de ponto flutuante que não está sufixada é de tipo `double`.
- A biblioteca de tratamento de sinal fornece a capacidade de registrar uma função para interceptar eventos inesperados com a função `signal`. A função `signal` recebe dois argumentos — um número de sinal inteiro e um ponteiro para a função de tratamento de sinal.

- Os sinais também podem ser gerados com a função `raise` e um argumento de inteiro.
- A biblioteca de utilitários gerais (`cstdlib`) fornece as funções `calloc` e `realloc` para alocação dinâmica de memória. Essas funções podem ser utilizadas para criar arrays dinâmicos.
- A função `calloc` recebe dois argumentos — o número de elementos (`nmemb`) e o tamanho de cada elemento (`size`) — e inicializa os elementos do array como zero. A função retorna um ponteiro para a memória alocada ou, se a memória não estiver alocada, a função retorna um ponteiro nulo.
- A função `realloc` muda o tamanho de um objeto alocado por uma chamada anterior a `malloc`, `calloc` ou `realloc`. O conteúdo do objeto original não é modificado, visto que a quantidade de memória alocada é maior do que a quantidade alocada anteriormente.
- A função `realloc` aceita dois argumentos — um ponteiro para o objeto original (`ptr`) e o novo tamanho do objeto (`size`). Se `ptr` for nulo, `realloc` funciona identicamente a `malloc`. Se `size` for 0 e o ponteiro recebido não for nulo, a memória do objeto é liberada. Caso contrário, se `ptr` não for nulo e `size` for maior que zero, `realloc` tenta alocar um novo bloco de memória para o objeto. Se o novo espaço não puder ser alocado, o objeto apontado por `ptr` permanecerá inalterado. A função `realloc` retorna um ponteiro para a memória realocada ou um ponteiro nulo.
- O resultado da instrução `goto` é uma alteração no fluxo de controle do programa. A execução do programa continua na primeira instrução depois do rótulo na instrução `goto`.
- Um rótulo é um identificador seguido por dois-pontos. Um rótulo deve aparecer na mesma função que a instrução `goto` que ele referencia.
- Uma `union` é um tipo de dados cujos membros compartilham o mesmo espaço de armazenamento. Os membros podem ser de quase todos os tipos. O armazenamento reservado para uma `union` é grande o bastante para armazenar seu maior membro. Na maioria dos casos, as `unions` contêm dois ou mais tipos de dados. Somente um membro e, portanto, um tipo de dados, pode ser referenciado por vez.
- Uma `union` é declarada no mesmo formato que uma estrutura.
- Uma `union` pode ser inicializada somente com um valor do tipo de seu primeiro membro ou outro objeto do mesmo tipo de união.
- O C++ permite ao programador fornecer especificações de linkagem para informar o compilador de que uma função foi compilada em um compilador C e impedir que o nome da função seja codificado pelo compilador C++.
- Para informar ao compilador que uma ou várias das funções foram compiladas em C, escreva os protótipos de função como mostrado a seguir:

```
extern "C" protótipo de função // uma única função
extern "C" // múltiplas funções
{
 protótipos de função
}
```

Essas declarações informam ao compilador que as funções especificadas não foram compiladas em C++, então a codificação de nome não deve ser realizada nas funções listadas na especificação de linkagem. Essas funções então podem ser vinculadas adequadamente com o programa.

- Os ambientes C++ normalmente incluem as bibliotecas C padrão e não requerem que o programador utilize especificações de linkagem para essas funções.

## Terminologia

>>, símbolo de acréscimo à saída	extern "C"	<i>pipe</i>
>, símbolo de redirecionamento de saída	extern, especificador de classe de	<i>pipng</i>
<, símbolo de redirecionamento de entrada	armazenamento	<i>raise</i>
<csignal>	float, sufixo (f ou F)	<i>realloc</i>
<csdarg>	free	redirecionamento de E/S
argumentos da linha de comando	instrução goto	<i>signal</i>
argv	instrução ilegal	<i>static</i> , especificador de classe de
arrays dinâmicos	interceptar	armazenamento
atexit	interrupção	<i>union</i>
biblioteca de tratamento de sinal	linkagem externa	<i>unsigned long</i> , sufixo de inteiro (ul ou UL)
calloc	linkagem interna	<i>unsigned</i> , sufixo de inteiro (u ou U)
const	lista de argumentos de comprimento variável	<i>va_arg</i>
evento	<i>long double</i> , sufixo (l ou L)	<i>va_end</i>
exceção de ponto flutuante	<i>long</i> , sufixo integer (l ou L)	<i>va_list</i>
exit	make	<i>va_start</i>
EXIT_FAILURE	Makefile	violação de segmentação
EXIT_SUCCESS	malloc	<i>volatile</i>

## Exercícios de revisão

**E.1** Preencha as lacunas em cada uma das seguintes sentenças:

- O símbolo \_\_\_\_\_ redireciona os dados de entrada a partir do teclado de modo que eles provenham de um arquivo.
- O símbolo \_\_\_\_\_ é utilizado para redirecionar a saída na tela para um arquivo.
- O símbolo \_\_\_\_\_ é utilizado para acrescentar a saída de um programa ao fim de um arquivo.
- Um \_\_\_\_\_ é utilizado para direcionar a saída de um programa como a entrada de outro programa.
- \_\_\_\_\_ na lista de parâmetros de uma função indicam que a função pode receber um número variável de argumentos.
- A macro \_\_\_\_\_ deve ser invocada antes que os argumentos em uma lista de argumentos de comprimento variável possam ser acessados.
- A macro \_\_\_\_\_ é utilizada para acessar os argumentos individuais de uma lista de argumentos de comprimento variável.
- A macro \_\_\_\_\_ realiza a faxina de terminação em uma função cuja lista de argumentos variável foi referenciada pela macro `va_start`.
- O argumento \_\_\_\_\_ de `main` recebe o número de argumentos em uma linha de comando.
- O argumento \_\_\_\_\_ de `main` armazena argumentos de linha de comando como strings de caracteres.
- O utilitário UNIX \_\_\_\_\_ lê um arquivo chamado \_\_\_\_\_ que contém instruções para compilar e vincular um programa consistindo em múltiplos arquivos-fonte. O utilitário recompila um arquivo somente se o arquivo (ou um cabeçalho que ele utiliza) tiver sido modificado desde a última vez que ele foi compilado.
- A função \_\_\_\_\_ força um programa a terminar a execução.
- A função \_\_\_\_\_ registra uma chamada de função na terminação normal do programa.
- Um(a) \_\_\_\_\_ do tipo inteiro ou de ponto flutuante pode ser acrescentado(a) a uma constante do tipo inteiro ou de ponto flutuante para especificar o tipo exato da constante.
- A função \_\_\_\_\_ pode ser utilizada a fim de registrar uma função para interceptar eventos inesperados.
- A função \_\_\_\_\_ gera um sinal a partir de dentro de um programa.
- A função \_\_\_\_\_ aloca dinamicamente a memória de um array e inicializa os elementos como zero.
- A função \_\_\_\_\_ altera o tamanho de um bloco de memória alocado dinamicamente.
- Um(a) \_\_\_\_\_ é uma entidade contendo uma coleção de variáveis que ocupa a mesma memória, mas em momentos diferentes.
- A palavra-chave \_\_\_\_\_ é utilizada para introduzir uma definição de união.

## Respostas dos exercícios de revisão

**E.1** a) `<` (redirecionamento de entrada). b) `>` (redirecionamento de saída). c) `>>` (acréscimo à saída). d) `pipe` (l). e) Reticências (...). f) `va_start`. g) `va_arg`. h) `va_end`. i) `argc`. j) `argv`. k) `make`, `Makefile` l) `exit`. m) `atexit`. n) sufixo. o) `signal`. p) `raise`. q) `calloc`. r) `realloc`. s) união. t) `union`.

## Exercícios

**E.2** Escreva um programa que calcula o produto de uma série de inteiros que são passados para a função `product` utilizando uma lista de argumentos de comprimento variável. Teste sua função com várias chamadas, cada uma com um número diferente de argumentos.

**E.3** Escreva um programa que imprime os argumentos de linha de comando do programa.

**E.4** Escreva um programa que classifica um array de inteiros em ordem crescente ou decrescente. O programa deve utilizar os argumentos de linha de comando para passar o argumento `-a` para ordem crescente ou `-d` para ordem decrescente. [Nota: Esse é o formato-padrão para passar opções para um programa em UNIX.]

**E.5** Leia os manuais do sistema para determinar quais sinais são suportados pela biblioteca de tratamento de sinal (`<csignal>`). Escreva um programa com handlers de sinal para os sinais `SIGABRT` e `SIGINT`. O programa deve testar a interceptação desses sinais chamando a função `abort` para gerar um sinal do tipo `SIGABRT` e pressionando `Ctrl+C` para gerar um sinal do tipo `SIGINT`.

**E.6** Escreva um programa que aloca dinamicamente um array de inteiros utilizando uma função a partir de `<cstdlib>`, não o operador `new`. O tamanho do array deve ser inserido a partir do teclado. Aos elementos do array devem ser atribuídos valores inseridos a partir do teclado. Imprima os valores do array. Em seguida, realoque a memória para o array para a metade do número atual de elementos. Imprima os valores que permanecem no array para confirmar se eles correspondem à primeira metade dos valores no array original.

**E.7** Escreva um programa que aceita dois nomes de arquivo como argumentos de linha de comando, lê um a um os caracteres do primeiro arquivo e os grava na ordem inversa ao segundo arquivo.

**E.8** Escreva um programa que utiliza instruções `goto` para simular uma estrutura de loop aninhada que imprime um quadrado de asteriscos como mostrado na Figura E.10. O programa deve utilizar somente as três instruções de saída a seguir:

```
cout << ' *';
cout << ' ';
cout << endl;
```

- E.9** Forneça a definição para union Data contendo char character1, short short1, long long1, float float1 e double double1.
- E.10** Crie uma union Integer com os membros char character1, short short1, int integer1 e long long1. Escreva um programa que insere valores do tipo char, short, int e long e os armazena em variáveis union do tipo union Integer. Cada variável union deve ser impressa como um char, um short, um int e um long. Os valores são sempre impressos corretamente?
- E.11** Crie uma union FloatingPoint com os membros float float1, double double1 e long double longDouble. Escreva um programa que insere valores do tipo float, double e long double e os armazena em variáveis union do tipo union FloatingPoint. Cada variável union deve ser impressa como um float, um double e um long double. Os valores são sempre impressos corretamente?
- E.12** Dada a union

```
union A
{
 double y;
 char *zPtr;
};
```

quais das seguintes instruções são corretas para inicializar a union?

- a) A p = b; // b é do tipo A
- b) A q = x; // x é um double
- c) A r = 3.14159;
- d) A s = { 79.63 };
- e) A t = { "Hi There!" };
- f) A u = { 3.14159, "Pi" };
- g) A v = { y = -7.843, zPtr = &x };

```

* *
* *
* *
* *

```

**Figura E.10** Exemplo para o Exercício E.8.

# F



*Zelai pelo bem; defini-o bem.*  
Alfred, Lord Tennyson

*Eu lhe dei uma argumentação;  
mas não sou obrigado a fazê-lo  
entendê-la.*  
Samuel Johnson

*Um bom símbolo é o melhor  
argumento e é um missionário  
para convencer milhares.*  
Ralph Waldo Emerson

*As condições são  
fundamentalmente favoráveis.*  
Herbert Hoover [dezembro de  
1929]

*A pessoa tendenciosa, quando  
engajada em uma disputa,  
em nada se preocupa com a  
verdade da questão, anseia  
apenas persuadir seus ouvintes  
de suas próprias convicções.*  
Platão

## Pré-processador

### OBJETIVOS

Neste apêndice, você aprenderá:

- A utilizar `#include` para desenvolver programas grandes.
- A utilizar `#define` para criar macros e macros com argumentos.
- O que é a compilação condicional.
- Como exibir mensagens de erro durante a compilação condicional.
- A utilizar assertivas para testar se os valores de expressões são corretos.

- F.1** Introdução
- F.2** A diretiva de pré-processor `#include`
- F.3** Diretiva de pré-processor `#define`: constantes simbólicas
- F.4** Diretiva de pré-processor `#define`: macros
- F.5** Compilação condicional
- F.6** As diretivas de pré-processor `#error` e `#pragma`
- F.7** Os operadores `#` e `##`
- F.8** Constantes simbólicas predefinidas
- F.9** Assertivas
- F.10** Síntese

Resumo | Terminologia | Exercícios de revisão | Respostas dos exercícios de revisão | Exercícios

## F.1 Introdução

Este apêndice oferece uma introdução ao **pré-processor**. O pré-processamento ocorre antes de um programa ser compilado. Algumas possíveis ações são a inclusão de outros arquivos no arquivo sendo compilado, a definição de **constantes simbólicas** e **macros**, a **compilação condicional** de código do programa e a **execução condicional de diretivas de pré-processor**. Todas as diretivas de pré-processor iniciam com `#`, e somente caracteres de espaço em branco podem aparecer antes de uma diretiva de pré-processor em uma linha. As diretivas de pré-processor não são instruções C++, então não acabam em um ponto-e-vírgula (`;`). As diretivas de pré-processor são processadas completamente antes de a compilação iniciar.



### Erro comum de programação F.1

*Colocar um ponto-e-vírgula no fim de uma diretiva de pré-processor pode levar a uma variedade de erros, dependendo do tipo de diretiva de pré-processor.*



### Observação de engenharia de software F.1

*Muitos recursos de pré-processor (especialmente macros) são mais apropriados a programadores em C do que a programadores em C++. Os programadores em C++ devem se familiarizar com o pré-processor, porque podem precisar trabalhar com o código legado C.*

## F.2 A diretiva de pré-processor `#include`

A **diretiva de pré-processor `#include`** foi utilizada por todo este texto. A diretiva `#include` faz com que uma cópia de um arquivo especificado seja incluída no lugar da diretiva. As duas formas da diretiva `#include` são

```
#include <filename>
#include "filename"
```

A diferença entre elas é a localização em que o pré-processor procura o arquivo a ser incluído. Se o nome de arquivo estiver entre colchetes angulares (`<` e `>`) — utilizados para arquivos de cabeçalho-padrão de biblioteca —, o pré-processor procura o arquivo especificado de maneira dependente da implementação, normalmente por diretórios pré-designados. Se o nome de arquivo estiver entre aspas, o pré-processor pesquisa primeiro o mesmo diretório enquanto o arquivo está sendo compilado e, então, da mesma maneira dependente da implementação, procura por um nome de arquivo entre colchetes angulares. Esse método é normalmente utilizado para incluir arquivos de cabeçalho definidos pelo programador.

A diretiva `#include` é utilizada para incluir arquivos de cabeçalho-padrão como `<iostream>` e `<iomanip>`. A diretiva `#include` também é utilizada com programas consistindo em vários arquivos-fonte que devem ser compilados juntos. Um arquivo de cabeçalho contendo declarações e definições comuns aos arquivos de programa separados é frequentemente criado e incluído no arquivo. Exemplos dessas declarações e definições são classes, estruturas, uniões, enumerações e protótipos de função, constantes e objetos de fluxo (por exemplo, `cin`).

## F.3 Diretiva de pré-processor `#define`: constantes simbólicas

A **diretiva de pré-processor `#define`** cria **constantes simbólicas** — constantes representadas como símbolos — e macros — operações definidas como símbolos. O formato da diretiva de pré-processor `#define` é

```
#define identificador texto substituto
```

Quando essa linha aparece em um arquivo, todas as ocorrências subseqüentes (exceto aquelas dentro de uma string) de *identificador* nesse arquivo serão substituídas pelo *texto substituto* antes de o programa ser compilado. Por exemplo,

```
#define PI 3.14159
```

substitui todas as ocorrências subseqüentes da constante simbólica PI pela constante numérica 3.14159. As constantes simbólicas permitem ao programador criar um nome para uma constante e utilizá-lo por todo o programa. Mais tarde, se a constante precisar ser modificada por todo o programa, ela pode ser modificada uma vez na diretiva de pré-processador #define — e quando o programa for recompilado, todas as ocorrências da constante no programa serão modificadas. [Nota: Tudo à direita do nome da constante simbólica substitui a constante simbólica. Por exemplo, #define PI = 3.14159 faz com que o pré-processador substitua cada ocorrência de PI por = 3.14159. Essa substituição é a causa de muitos erros lógicos sutis e erros de sintaxe.] Redefinir uma constante simbólica com um novo valor sem primeiro defini-la também é um erro. Observe que as variáveis const em C++ são preferidas às constantes simbólicas. As variáveis constantes têm um tipo de dados específico e são visíveis pelo nome a um depurador. Uma vez que uma constante simbólica é substituída por seu texto substituto, somente o texto substituto é visível a um depurador. Uma desvantagem das variáveis const é que elas podem requerer uma posição de memória de seus tamanhos de tipos de dados — constantes simbólicas não requerem nenhuma memória adicional.



### Erro comum de programação F.2

Utilizar constantes simbólicas em um arquivo diferente do arquivo em que as constantes simbólicas são definidas é um erro de compilação (a menos que elas estejam incluídas com #include a partir de um arquivo de cabeçalho).



### Boa prática de programação F.1

Utilizar nomes significativos para constantes simbólicas ajuda a tornar os programas mais autodocumentados.

## F.4 Diretiva de pré-processador #define: macros

[Nota: Esta seção foi incluída para o benefício de programadores em C++ que precisarão trabalhar com o código legado C. Em C++, as macros podem ser frequentemente substituídas por templates e funções inline.] Uma macro é uma operação definida em uma diretiva de pré-processador #define. Como com as constantes simbólicas, o *identificador de macro* é substituído pelo *texto substituto* antes de o programa ser compilado. As macros podem ser definidas com ou sem *argumentos*. Uma macro sem argumentos é processada como uma constante simbólica. Em uma macro com argumentos, os argumentos são substituídos no texto substituto, depois a macro é expandida — isto é, o texto substituto substitui o identificador de macro e a lista de argumentos no programa. [Nota: Não há nenhuma verificação de tipo de dados para argumentos de macro. Uma macro é utilizada simplesmente para substituição de texto.]

Considere a seguinte definição de macro com um argumento para a área de um círculo:

```
#define CIRCLE_AREA(x) (PI * (x) * (x))
```

Onde quer que CIRCLE\_AREA( y ) apareça no arquivo, o valor de y é substituído por x no texto substituto, a constante simbólica PI é substituída por seu valor (definido previamente) e a macro é expandida no programa. Por exemplo, a instrução

```
area = CIRCLE_AREA(4);
```

é expandida para

```
area = (3.14159 * (4) * (4));
```

Como a expressão consiste somente em constantes, o valor da expressão em tempo de compilação pode ser avaliado e o resultado é atribuído à area em tempo de execução. Os parênteses em torno de cada x no texto substituto e em torno da expressão inteira forçam a ordem adequada de avaliação quando o argumento de macro for uma expressão. Por exemplo, a instrução

```
area = CIRCLE_AREA(c + 2);
```

é expandida para

```
area = (3.14159 * (c + 2) * (c + 2));
```

que avalia corretamente, porque os parênteses forçam a ordem adequada de avaliação. Se os parênteses são omitidos, a expansão da macro é

```
area = 3.14159 * c + 2 * c + 2;
```

avalia incorretamente como

```
area = (3.14159 * c) + (2 * c) + 2;
```

por causa das regras de precedência de operadores.



### Erro comum de programação F.3

Esquecer de colocar argumentos de macro entre parênteses no texto substituto é um erro.



A macro `CIRCLE_AREA` poderia ser definida como uma função. A função `circleArea`, como em

```
double circleArea(double x) { return 3.14159 * x * x; }
```

realiza o mesmo cálculo que `CIRCLE_AREA`, mas o overhead de uma chamada de função é associado com a função `circleArea`. As vantagens de `CIRCLE_AREA` são que as macros inserem o código diretamente no programa — evitando o overhead de função — e que o programa permanece legível porque `CIRCLE_AREA` é definida separadamente e nomeada de modo significativo. Uma desvantagem é que seu argumento é avaliado duas vezes. Além disso, toda vez que uma macro aparece em um programa, a macro é expandida. Se a macro for grande, isso aumenta o tamanho do programa. Portanto, há uma compensação entre velocidade de execução e tamanho de programa (se o espaço em disco for pequeno). Observe que as funções `inline` (ver Capítulo 3) são preferidas para obter o desempenho de macros e os benefícios de engenharia de software de funções.



### Dica de desempenho F.1

*Às vezes as macros podem ser utilizadas para substituir uma chamada de função pelo código `inline` antes do tempo de execução. Isso elimina o overhead de uma chamada de função. As funções `inline` são preferíveis às macros porque oferecem os serviços de verificação dos tipos de funções.*

O seguinte é uma definição de macro com dois argumentos para a área de um retângulo:

```
#define RECTANGLE_AREA(x, y) ((x) * (y))
```

Onde quer que `RECTANGLE_AREA( a, b )` apareça no programa, os valores de `a` e `b` são substituídos no texto substituto de macro, e a macro é expandida no lugar do nome de macro. Por exemplo, a instrução

```
rectArea = RECTANGLE_AREA(a + 4, b + 7);
```

é expandida para

```
rectArea = ((a + 4) * (b + 7));
```

O valor da expressão é avaliado e atribuído à variável `rectArea`.

O texto substituto para uma macro ou constante simbólica é normalmente qualquer texto na linha depois do identificador na diretiva `#define`. Se o texto substituto de uma macro ou de uma constante simbólica for mais longo do que o restante da linha, uma barra invertida (`\`) deve ser colocada no fim de cada linha da macro (exceto na última linha), indicando que o texto substituto continua na próxima linha.

As constantes simbólicas e as macros podem ser descartadas utilizando a **diretiva de pré-processador `#undef`**. A diretiva `#undef` ‘`undefine`’ um nome de constante simbólica ou de macro. O escopo de uma constante simbólica ou macro é a partir de sua definição até que ele seja indefinido com `#undef` ou até que o fim do arquivo seja alcançado. Uma vez indefinido, um nome pode ser redefinido com `#define`.

Observe que as expressões com efeitos colaterais (por exemplo, valores de variáveis são modificados) não devem ser passadas para uma macro, porque os argumentos de macro podem ser avaliados mais de uma vez.



### Erro comum de programação F.4

*Muitas vezes as macros acabam substituindo um nome que não se destinava a ser utilizado pela macro, mas que casualmente foi digitado da mesma forma. Isso pode resultar em erros de sintaxe e compilação excepcionalmente misteriosos.*

## F.5 Compilação condicional

A **compilação condicional** permite ao programador controlar a execução de diretivas de pré-processador e a compilação do código do programa. Cada uma das diretivas condicionais de pré-processador avalia uma expressão que resulta em uma constante do tipo inteiro que determinará se o código será compilado. Expressões de coerção, expressões `sizeof` e constantes de enumeração não podem ser avaliadas em diretivas de pré-processador porque todas essas são determinadas pelo compilador e o pré-processamento acontece antes da compilação.

A construção condicional de pré-processador é muito semelhante à estrutura de seleção `if`. Considere o seguinte código de pré-processador:

```
#ifndef NULL
#define NULL 0
#endif
```

Essas diretivas determinam se a constante simbólica `NULL` já está definida. A expressão `#ifndef NULL` inclui o código até `#endif` se `NULL` não estiver definido e pula o código se `NULL` estiver definido. Cada construção `#if` termina com `#endif`. As diretivas **`#ifdef`** e **`#ifndef`** são abreviadas para `#if defined( nome )` e `#if !defined( nome )`. Uma construção de pré-processador condicional de múltiplas partes pode ser testada utilizando as diretivas `#elif` (a equivalente de `else if` em uma estrutura `if`) e a `#else` (a equivalente de `else` em uma estrutura `if`).

Durante o desenvolvimento do programa, os programadores acham útil ‘desativar’ grandes partes de código para impedir que ele seja compilado. Se o código contém comentários no estilo C, o `/*` e o `*/` não podem ser utilizados para realizar essa tarefa, porque o primeiro `*/` encontrado terminaria o comentário. Em vez disso, o programador pode utilizar a seguinte construção de pré-processador:

```
#if 0
 código cancelado pela compilação
#endif
```

Para permitir ao código ser compilado, simplesmente substitua o valor 0 na construção precedente pelo valor 1.

A compilação condicional é comumente utilizada como um recurso auxiliar de depuração. Instruções de saída costumam ser utilizadas para imprimir valores de variáveis e confirmar o fluxo de controle. Essas instruções de saída podem ser incluídas em diretivas de pré-processador condicionais para que as instruções sejam compiladas somente até que o processo de depuração seja concluído. Por exemplo,

```
#ifndef DEBUG
 cerr << "Variable x = " << x << endl;
#endif
```

faz com que a instrução `cerr` seja compilada no programa se a constante simbólica `DEBUG` tiver sido definida antes da diretiva `#ifndef` `DEBUG`. Esta constante simbólica é normalmente configurada por um compilador de linha de comando ou por configurações no IDE (por exemplo, Visual Studio .NET) e não por uma definição `#define` explícita. Quando a depuração estiver completada, a diretiva `#define` é removida do arquivo-fonte e as instruções de saída inseridas para propósitos de depuração são ignoradas durante a compilação. Em programas maiores, poderia ser desejável definir várias constantes simbólicas diferentes que controlem a compilação condicional em seções separadas do arquivo-fonte.



### Erro comum de programação F.5

*Inserir instruções de saída compiladas condicionalmente para propósitos de depuração em localizações em que o C++ atualmente espera uma única instrução pode levar a erros de sintaxe e de lógica. Nesse caso, a instrução condicionalmente compilada deve ser incluída em uma instrução composta. Portanto, quando o programa é compilado com instruções de depuração, o fluxo de controle do programa não é alterado.*

## F.6 As diretivas de pré-processador `#error` e `#pragma`

A diretiva `#error`

`#error` tokens

imprime uma mensagem dependente de implementação incluindo os *tokens* especificados na diretiva. Os tokens são seqüências de caracteres separados por espaços. Por exemplo,

```
#error 1 - Out of range error
```

contém seis tokens. Em um compilador C++ popular, por exemplo, quando uma diretiva `#error` é processada, os tokens na diretiva são exibidos como uma mensagem de erro, o pré-processamento pára e o programa não compila.

A diretiva `#pragma`

`#pragma` tokens

causa uma ação definida pela implementação. Um `pragma` não reconhecido pela implementação é ignorado. Um compilador C++ particular, por exemplo, poderia reconhecer `pragmas` que permitem ao programador tirar proveito de capacidades específicas desse compilador. Para obter informações adicionais sobre `#error` e `#pragma`, veja a documentação da sua implementação do C++.

## F.7 Os operadores `#` e `##`

Os operadores de pré-processador `#` e `##` estão disponíveis em C++ e ANSI/ISO C. O operador `#` faz com que um token de texto para substituição seja convertido em uma string entre aspas. Considere a seguinte definição de macro:

```
#define HELLO(x) cout << "Hello, " << x << endl;
```

Quando `HELLO(John)` aparece em um arquivo de programa, ela é expandida para

```
cout << "Hello, " << "John" << endl;
```

A string `"John"` substitui `x` no texto substituído. As strings separadas por espaço em branco são concatenadas durante o pré-processamento, então a instrução acima é equivalente a

```
cout << "Hello, John" << endl;
```

Observe que o operador `#` deve ser utilizado em uma macro com argumentos, porque o operando de `#` refere-se a um argumento da macro.

O operador `##` concatena dois tokens. Considere a seguinte definição de macro:

```
#define TOKENCONCAT(x, y) x ## y
```

Quando `TOKENCONCAT` aparece no programa, seus argumentos são concatenados e utilizados para substituir a macro. Por exemplo, `TOKENCONCAT( 0, K )` é substituído por `0K` no programa. O operador `##` deve ter dois operandos.

## F.8 Constantes simbólicas predefinidas

Há seis **constantes simbólicas predefinidas** (Figura F.1). Os identificadores para cada um desses começam (e, exceto por `__cplusplus`, terminam), com *dois* sublinhados. Esses identificadores e o operador de pré-processor `defined` (Seção F.5) não podem ser utilizados na diretiva `#define` ou `#undefs`.

## F.9 Assertivas

A **macro `assert`** — definida no arquivo de cabeçalho `<cassert>` — testa o valor de uma expressão. Se o valor da expressão for 0 (falso), então `assert` imprime uma mensagem de erro e chama a função `abort` (da biblioteca de utilitários gerais — `<cstdlib>`) para terminar a execução do programa. Essa é uma ferramenta de depuração útil para testar se uma variável tem ou não um valor correto. Por exemplo, suponha que a variável `x` nunca deva ser maior que 10 em um programa. Uma assertiva pode ser utilizada para testar o valor de `x` e imprimir uma mensagem de erro se o valor de `x` for incorreto. A instrução seria

```
assert(x <= 10);
```

Se `x` for maior que 10 quando a instrução anterior for encontrada em um programa, uma mensagem de erro contendo o número da linha e o nome de arquivo será impressa e o programa terminará. O programador pode então se concentrar nessa área do código para localizar o erro. Se a constante simbólica `NDEBUG` estiver definida, as assertivas subsequentes serão ignoradas. Portanto, quando as assertivas não forem mais necessárias (isto é, quando a depuração estiver completa), inserimos a linha no arquivo do programa em vez de excluir cada assertiva manualmente. Como com a constante simbólica `DEBUG`, `NDEBUG` é frequentemente configurada por opções de linha de comando do compilador ou por uma configuração no IDE.

```
#define NDEBUG
```

A maioria dos compiladores C++ agora inclui o tratamento de exceções. Os programadores em C++ preferem utilizar exceções em vez de assertivas. Mas as assertivas ainda são valiosas para programadores em C++ que trabalham com código C de legado.

## F.10 Síntese

Este apêndice discuti a diretiva `#include`, que é utilizada para desenvolver programas maiores. Você também aprendeu sobre a diretiva `#define`, que é utilizada para criar macros. Introduzimos a compilação condicional, exibindo mensagens de erro e utilizando assertivas. No próximo apêndice, você implementará o design do sistema ATM a partir do “Estudo de caso de engenharia de software” localizado nos capítulos 1–7, 9 e 13.

Constante simbólica	Descrição
<code>__LINE__</code>	O número da linha do código-fonte atual (uma constante do tipo inteiro).
<code>__FILE__</code>	O nome presumido do arquivo-fonte (uma string).
<code>__DATE__</code>	A data em que o arquivo-fonte é compilado (uma string no formato " <b>Mmm dd yyyy</b> " como " <b>Aug 19 2002</b> ").
<code>__STDC__</code>	Indica se o programa obedece ao padrão ANSI/ISO C. Contém o valor 1 se há completa obediência; caso contrário, é indefinido.
<code>__TIME__</code>	A hora em que o arquivo-fonte é compilado (um literal string na forma " <b>hh:mm:ss</b> ").
<code>__cplusplus</code>	Contém o valor 199711L (a data em que o padrão ISO C++ foi aprovado) se o arquivo estiver sendo compilado por um compilador C++; caso contrário, indefinido. Permite a um arquivo ser configurado para ser compilado como C ou C++.

**Figura F.1** As constantes simbólicas predefinidas.

## Resumo

- Todas as diretivas de pré-processador iniciam com # e são processadas antes de o programa ser compilado.
- Somente os caracteres de espaço em branco podem aparecer antes de uma diretiva de pré-processador em uma linha.
- A diretiva `#include` inclui uma cópia do arquivo especificado. Se o nome do arquivo estiver entre aspas, o pré-processador começa a pesquisar o arquivo a ser incluído no mesmo diretório que o arquivo sendo compilado. Se o nome do arquivo estiver entre colchetes angulares (`<` e `>`), a pesquisa é realizada de maneira definida pela implementação.
- A diretiva de pré-processador `#define` é utilizada para criar constantes simbólicas e macros.
- Uma constante simbólica é um nome para uma constante.
- Uma macro é uma operação definida em uma diretiva de pré-processador `#define`. As macros podem ser definidas com ou sem argumentos.
- O texto substituto para uma macro ou constante simbólica é qualquer texto que permanece na linha depois do identificador (e, se houver, a lista de argumentos de macro) na diretiva `#define`. Se o texto substituto para uma macro ou constante simbólica for muito longo para se ajustar em uma linha, uma barra invertida (`\`) é colocada no fim da linha, indicando que o texto substituto continua na próxima linha.
- Constantes simbólicas e macros podem ser descartadas utilizando a diretiva de pré-processador `#undef`. A diretiva `#undef` 'undefine' o nome da constante simbólica ou macro.
- O escopo de uma constante simbólica ou macro é a partir de sua definição até que ele seja indefinido com `#undef` ou até que o fim do arquivo seja alcançado.
- A compilação condicional permite ao programador controlar a execução de diretivas de pré-processador e a compilação do código do programa.
- As diretivas de pré-processador condicionais avaliam expressões que resultam em constantes do tipo inteiro. Expressões de coerção, expressões `sizeof` e constantes de enumeração não podem ser avaliadas em diretivas de pré-processador.
- Cada construção `#if` termina com `#endif`.
- As diretivas `#ifdef` e `#ifndef` são fornecidas como abreviação para `#if defined(nome)` e `#if ! defined(nome)`.
- Uma construção condicional de pré-processador de múltiplas partes é testada com diretivas `#elif` e `#else`.
- A diretiva `#error` imprime uma mensagem dependente de implementação que inclui os tokens especificados na diretiva e termina o pré-processamento e a compilação.
- A diretiva `#pragma` produz uma ação definida pela implementação. Se o `pragma` não for reconhecido pela implementação, ele é ignorado.
- O operador `#` faz com que o token de texto substituto seguinte seja convertido em uma string entre aspas. O operador `#` deve ser utilizado em uma macro com argumentos, porque o operando de `#` deve ser um argumento da macro.
- O operador `##` concatena dois tokens. O operador `##` deve ter dois operandos.
- Há seis constantes simbólicas predefinidas. A constante `__LINE__` é o número da linha do código-fonte atual (um inteiro). A constante `__FILE__` é o nome presumido do arquivo (uma string). A constante `__DATE__` é a data em que o arquivo-fonte foi compilado (uma string). A constante `__TIME__` é a hora em que o arquivo-fonte foi compilado (uma string). Observe que cada uma das constantes simbólicas predefinidas inicia (e, com a exceção de `__cplusplus`, termina) com dois sublinhados.
- A macro `assert` — definida no arquivo de cabeçalho `<cassert>` — testa o valor de uma expressão. Se o valor da expressão for 0 (falso), então `assert` imprime uma mensagem de erro e chama a função `abort` para terminar a execução do programa.

## Terminologia

<code>##</code> , operador de pré-processador de concatenação	<code>__DATE__</code>	constantes simbólicas
<code>#define</code>	<code>__FILE__</code>	constantes simbólicas predefinidas
<code>#elif</code>	<code>__LINE__</code>	convert-to-string, diretiva de pré-processador
<code>#else</code>	<code>__TIME__</code>	depurador
<code>#endif</code>	<code>&lt;cassert&gt;</code>	diretiva de pré-processamento
<code>#error</code>	<code>&lt;stdio&gt;</code>	diretivas
<code>#if</code>	<code>&lt;stdlib&gt;</code>	escopo de uma constante simbólica ou macro
<code>#ifdef</code>	<code>abort</code>	execução condicional de pré-processador
<code>#ifndef</code>	argumento	expandir uma macro
<code>#include "filename"</code>	arquivo de cabeçalho	macro
<code>#include &lt;filename&gt;</code>	arquivos de cabeçalho da biblioteca-padrão	macro com argumentos
<code>#pragma</code>	<code>assert</code>	operador <code>#</code>
<code>#undef</code>	caractere de continuação <code>\</code> (barra invertida)	pré-processador
<code>__cplusplus</code>	compilação condicional	texto substituto

## Exercícios de revisão

- F.1** Preencha as lacunas em cada uma das seguintes sentenças:
- Cada diretiva de pré-processor deve iniciar com \_\_\_\_\_.
  - A construção de compilação condicional pode ser estendida para testar múltiplos casos utilizando as diretivas \_\_\_\_\_ e \_\_\_\_\_.
  - A diretiva \_\_\_\_\_ cria macros e constantes simbólicas.
  - Somente os caracteres \_\_\_\_\_ podem aparecer antes de uma diretiva de pré-processor em uma linha.
  - A diretiva \_\_\_\_\_ descarta nomes de constantes simbólicas e de macro.
  - As diretivas \_\_\_\_\_ e \_\_\_\_\_ são fornecidas como notação abreviada para `#if defined(nome)` e `#if !defined(nome)`.
  - A \_\_\_\_\_ permite ao programador controlar a execução de diretivas de pré-processor e a compilação de código do programa.
  - A macro \_\_\_\_\_ imprime uma mensagem e termina a execução do programa se o valor da expressão que a macro avalia for 0.
  - A diretiva \_\_\_\_\_ insere um arquivo em outro arquivo.
  - O operador \_\_\_\_\_ concatena seus dois argumentos.
  - O operador \_\_\_\_\_ converte seu operando em uma string.
  - O caractere \_\_\_\_\_ indica que o texto substituto para uma constante simbólica ou macro continua na próxima linha.
- F.2** Escreva um programa para imprimir os valores das constantes simbólicas predefinidas `__LINE__`, `__FILE__`, `__DATE__` e `__TIME__` listadas na Figura F.1.
- F.3** Escreva uma diretiva de pré-processor para realizar cada uma das seguintes operações:
- Defina as constantes simbólicas YES para ter o valor 1.
  - Defina as constantes simbólicas NO para ter o valor 0.
  - Inclua o arquivo de cabeçalho `common.h`. O cabeçalho encontra-se no mesmo diretório que o arquivo sendo compilado.
  - Se a constante simbólica TRUE for definida, você deve indefini-la e redefini-la como 1. Não utilize `#ifdef`.
  - Se a constante simbólica TRUE for definida, você deve indefini-la e redefini-la como 1. Utilize a diretiva de pré-processor `#ifdef`.
  - Se a constante simbólica ACTIVE não for igual a 0, defina a constante simbólica INACTIVE como 0. Caso contrário, defina INACTIVE como 1.
  - Defina a macro `CUBE_VOLUME` que calcula o volume de um cubo (aceita um argumento).

## Respostas dos exercícios de revisão

- F.1** a) #. b) `#elif`, `#else`. c) `#define`. d) espaço em branco. e) `#undef`. f) `#ifdef`, `#ifndef`. g) compilação condicional. h) `assert`. i) `#include`. j) `##`. k) #. l) \.
- F.2** (Ver abaixo.)

```

1 // exF_02.cpp
2 // Solução do Exercício F.2 da revisão.
3 #include <iostream>
4
5 using std::cout;
6 using std::endl;
7
8 int main()
9 {
10 cout << "__LINE__ = " << __LINE__ << endl
11 << "__FILE__ = " << __FILE__ << endl
12 << "__DATE__ = " << __DATE__ << endl
13 << "__TIME__ = " << __TIME__ << endl
14 << "__cplusplus = " << __cplusplus << endl;
15
16 return 0;
17
18 } // fim do main

```

```

__LINE__ = 9
__FILE__ = c:\cpp4e\ch19\ex19_02.CPP
__DATE__ = Jul 17 2002
__TIME__ = 09:55:58
__cplusplus = 199711L

```

- F.3**
- a) `#define YES 1`
  - b) `#define NO 0`
  - c) `#include "common.h"`
  - d) `#if defined(TRUE)`  
`#undef TRUE`  
`#define TRUE 1`  
`#endif`
  - e) `#ifdef TRUE`  
`#undef TRUE`  
`#define TRUE 1`  
`#endif`
  - f) `#if ACTIVE`  
`#define INACTIVE 0`  
`#else`  
`#define INACTIVE 1`  
`#endif`
  - g) `#define CUBE_VOLUME( x ) ( ( x ) * ( x ) * ( x ) )`

## Exercícios

- F.4** Escreva um programa que defina uma macro com um argumento para calcular o volume de uma esfera. O programa deve calcular o volume de esferas de raios de 1 a 10 e imprimir os resultados em formato tabular. A fórmula para o volume de uma esfera é
- $$\left( \frac{4.0}{3} \right) * \pi * r^3$$
- onde  $\pi$  é 3.14159.

- F.5** Escreva um programa que produza a seguinte saída:

```
The sum of x and y is 13
```

O programa deve definir a macro SUM com dois argumentos, x e y, e utilizar SUM para produzir a saída.

- F.6** Escreva um programa que utiliza a macro MINIMUM2 para determinar o menor de dois valores numéricos. Insira os valores do teclado.
- F.7** Escreva um programa que utiliza a macro MINIMUM3 para determinar o menor de três valores numéricos. A macro MINIMUM3 deve utilizar a macro MINIMUM2 definida no Exercício F.6 para determinar o menor número. Insira os valores do teclado.
- F.8** Escreva um programa que utiliza a macro PRINT para imprimir um valor de string.
- F.9** Escreva um programa que utiliza a macro PRINTARRAY para imprimir um array de inteiros. A macro deve receber o array e o número de elementos no array como argumentos.
- F.10** Escreva um programa que utiliza a macro SUMARRAY para somar os valores em um array numérico. A macro deve receber o array e o número de elementos no array como argumentos.
- F.11** Reescreva as soluções dos exercícios F.4. a F.10 como funções inline.
- F.12** Para cada uma das macros a seguir, identifique os possíveis problemas (se houver algum) quando o pré-processador expandir as macros:
- a) `#define SQR( x ) x * x`
  - b) `#define SQR( x ) ( x * x )`
  - c) `#define SQR( x ) ( x ) * ( x )`
  - d) `#define SQR( x ) ( ( x ) * ( x ) )`

---



# Código para o estudo de caso do ATM

## G.1 Implementação do estudo de caso ATM

Este apêndice contém a implementação funcional completa do sistema ATM que projetamos nas seções “Estudo de caso de engenharia de software” localizadas no final dos capítulos 1–7, 9 e 13. A implementação abrange 877 linhas de código C++. Consideramos as classes na ordem em que as identificamos na Seção 3.11:

- ATM
- Screen
- Keypad
- CashDispenser
- DepositSlot
- Account
- BankDatabase
- Transaction
- BalanceInquiry
- Withdrawal
- Deposit

Aplicamos as diretrizes discutidas na Seção 9.12 e na Seção 13.10 para codificar essas classes com base na maneira como as modelamos nos diagramas de classes da UML das figuras 13.28 e 13.29. Para desenvolver as definições de funções-membro das classes, recorremos aos diagramas de atividades apresentados na Seção 5.11 e aos diagramas de comunicação e sequência apresentados na Seção 7.12. Observe que o nosso projeto do ATM não especifica toda a lógica do programa e talvez não especifique todos os atributos e operações necessários para completar a implementação do ATM. Isso é uma parte normal do processo de um projeto orientado a objetos. À medida que implementarmos o sistema, completaremos a lógica do programa e adicionaremos atributos e comportamentos conforme necessário para construir o sistema ATM especificado pelo documento de requisitos na Seção 2.8.

Concluimos a discussão apresentando um programa C++ (`ATMCaseStudy.cpp`) que inicia o ATM e coloca em uso as outras classes no sistema. Lembre-se de que estamos desenvolvendo uma primeira versão do sistema ATM que é executada em um computador pessoal e utiliza o teclado e o monitor do computador para simular o teclado numérico e a tela do sistema ATM. Também apenas simulamos as ações do dispensador de cédulas e da abertura para depósito. Tentamos, porém, implementar o sistema de modo que versões reais de hardware desses dispositivos possam ser integradas sem alterações significativas no código.



## G.2 Classe ATM

A classe ATM (figuras G.1–G.2) representa o ATM como um todo. A Figura G.1 contém a definição da classe ATM, incluída nas diretivas de pré-processador `#ifndef`, `#define` e `#endif` para assegurar que essa definição seja incluída apenas uma vez em um programa. Discutiremos as linhas 6–11 em breve. As linhas 16–17 contêm os protótipos de função para as funções-membro `public` da classe. O diagrama de classes da Figura 13.29 não lista nenhuma operação para a classe ATM, mas agora declaramos uma função-membro `public` `run` (linha 17) na classe ATM que permite a um cliente externo da classe (isto é, `ATMCaseStudy.cpp`) instruir o ATM para executar. Incluímos também um protótipo de função para um construtor-padrão (linha 16), que discutiremos em seguida.

As linhas 19–25 da Figura G.1 implementam os atributos da classe como membros de dados `private`. Determinamos todos esses atributos, exceto um, a partir dos diagramas de classes da UML das figuras 13.28 e 13.29. Observe que implementamos o atributo UML `Boolean` `userAuthenticated` na Figura 13.29 como um membro de dados `bool` em C++ (linha 19). A linha 20 declara um membro de dados não localizado em nosso design UML — um membro de dados `int` `currentAccountNumber` que monitora o número de conta do usuário autenticado atual. Logo veremos como a classe utiliza esse membro de dados.

As linhas 21–24 criam objetos para representar as partes do ATM. Considerando o diagrama de classes da Figura 13.28, lembre-se de que a classe ATM tem relacionamentos de composição com as classes `Screen`, `Keypad`, `CashDispenser` e `DepositSlot`, portanto a classe ATM é responsável pela criação delas. A linha 25 cria um `BankDatabase`, com o qual o ATM interage para acessar e manipular informações de conta bancária. [Nota: Se esse sistema ATM fosse real, a classe ATM receberia uma referência a um objeto existente de banco de dados criado pelo banco. Entretanto, nessa implementação só estamos simulando o banco de dados do banco, portanto a classe

```

1 // ATM.h
2 // Definição da classe ATM. Representa um caixa automático.
3 #ifndef ATM_H
4 #define ATM_H
5
6 #include "Screen.h" // definição da classe Screen
7 #include "Keypad.h" // definição da classe Keypad
8 #include "CashDispenser.h" // definição da classe CashDispenser
9 #include "DepositSlot.h" // definição da classe DepositSlot
10 #include "BankDatabase.h" // definição da classe BankDatabase
11 class Transaction; // declaração forward da classe Transaction
12
13 class ATM
14 {
15 public:
16 ATM(); // o construtor inicializa membros de dados
17 void run(); // inicia o ATM
18 private:
19 bool userAuthenticated; // se o usuário foi autenticado
20 int currentAccountNumber; // número atual da conta de usuário
21 Screen screen; // tela do ATM
22 Keypad keypad; // teclado do ATM
23 CashDispenser cashDispenser; // dispensador de cédulas do ATM
24 DepositSlot depositSlot; // abertura para depósito do ATM
25 BankDatabase bankDatabase; // banco de dados com as informações sobre as contas
26
27 // funções utilitárias private
28 void authenticateUser(); // tenta autenticar o usuário
29 void performTransactions(); // realiza transações
30 int displayMainMenu() const; // exibe o menu principal
31
32 // retorna o objeto da classe derivada Transaction especificada
33 Transaction *createTransaction(int);
34 }; // fim da classe ATM
35
36 #endif // ATM_H

```

**Figura G.1** Definição da classe ATM, que representa o ATM.

ATM cria o objeto BankDatabase com o qual interage.] Observe que as linhas 6–10 incluem (`#include`) as definições das classes Screen, Keypad, CashDispenser, DepositSlot e BankDatabase para que o ATM possa armazenar objetos dessas classes.

As linhas 28–30 e 33 contêm protótipos de função para as funções utilitárias `private` que a classe utiliza para realizar suas tarefas. Veremos logo como essas funções servem à classe. Observe que a função-membro `createTransaction` (linha 33) retorna um ponteiro `Transaction`. Para incluir o nome da classe `Transaction` nesse arquivo, devemos pelo menos incluir uma declaração antecipada (*forward*) da classe `Transaction` (linha 11). Lembre-se de que uma declaração antecipada informa ao compilador que existe uma classe, mas que a classe é definida em outra parte. Uma declaração antecipada é suficiente aqui, já que estamos utilizando somente `Transaction` como um tipo de retorno — se estivéssemos criando um objeto `Transaction` real, precisaríamos incluir (`#include`) todo o arquivo de cabeçalho `Transaction`.

### Definições de função-membro da classe **ATM**

A Figura G.2 contém as definições de função-membro da classe `ATM`. As linhas 3–7 incluem (`#include`) os arquivos de cabeçalho requeridos pelo arquivo de implementação `ATM.cpp`. Observe que incluir o arquivo de cabeçalho `ATM` permite ao compilador assegurar que as funções-membro da classe estão definidas corretamente. Isso também permite às funções-membro utilizar os membros de dados da classe.

A linha 10 declara um `enum` chamado `MenuOption` que contém constantes correspondentes às quatro opções no menu principal do ATM (isto é, consulta de saldo, retirada, depósito e saída). Observe que configurar `BALANCE_INQUIRY` como 1 faz com que as constantes de enumeração subsequentes sejam atribuídos os valores 2, 3 e 4, como valores constantes de enumeração incrementados por 1.

As linhas 13–18 definem o construtor da classe `ATM` que inicializa os membros de dados da classe. Quando um objeto `ATM` é inicialmente criado, nenhum usuário está autenticado, assim a linha 14 utiliza um inicializador de membro para configurar `userAuthenticated` como `false`. De maneira semelhante, a linha 15 inicializa `currentAccountNumber` como 0 porque ainda não há nenhum usuário atual.

A função-membro `ATM::run` (linhas 21–38) utiliza um loop infinito (linhas 24–37) para repetidamente dar boas-vindas a um usuário, tentar autenticar o usuário e, se a autenticação for bem-sucedida, permitir ao usuário realizar a transação. Depois que um usuário autenticado realiza as transações desejadas e escolhe sair, o ATM redefine a si mesmo, exibe uma mensagem de adeus para o usuário e reinicia o processo. Aqui, utilizamos um loop infinito para simular o fato de que um ATM parece executar continuamente até o banco desligá-lo (uma ação além do controle do usuário). Um usuário do ATM tem a opção de sair do sistema, mas não tem a capacidade de desligar o ATM completamente.

```

1 // ATM.cpp
2 // Definições de função-membro para a classe ATM.
3 #include "ATM.h" // definição da classe ATM
4 #include "Transaction.h" // definição da classe Transaction
5 #include "BalanceInquiry.h" // definição da classe BalanceInquiry
6 #include "Withdrawal.h" // definição da classe Withdrawal
7 #include "Deposit.h" // definição da classe Deposit
8
9 // constantes de enumeração representam as principais opções de menu
10 enum MenuOption { BALANCE_INQUIRY = 1, WITHDRAWAL, DEPOSIT, EXIT };
11
12 // o construtor-padrão ATM inicializa os membros de dados
13 ATM::ATM()
14 : userAuthenticated (false), // o usuário não foi autenticado para iniciar
15 currentAccountNumber(0) // nenhum número atual de conta para iniciar
16 {
17 // corpo vazio
18 } // fim do construtor-padrão ATM
19
20 // inicia o ATM
21 void ATM::run()
22 {
23 // dá boas-vindas e autentica o usuário; realiza transações
24 while (true)
25 {
26 // faz um loop enquanto o usuário ainda não está autenticado
27 while (!userAuthenticated)
28 {

```

**Figura G.2** Definições de função-membro da classe `ATM`.

(continua)

```

29 screen.displayMessageLine("\nWelcome!");
30 authenticateUser(); // autentica o usuário
31 } // fim do while
32
33 performTransactions(); // o usuário agora está autenticado
34 userAuthenticated = false; // reinicializa antes da próxima sessão do ATM
35 currentAccountNumber = 0; // reinicializa antes da próxima sessão do ATM
36 screen.displayMessageLine("\nThank you! Goodbye!");
37 } // fim do while
38 } // fim da função run
39
40 // tenta autenticar o usuário contra o banco de dados
41 void ATM::authenticateUser()
42 {
43 screen.displayMessage("\nPlease enter your account number: ");
44 int accountNumber = keypad.getInput(); // insere o número de conta
45 screen.displayMessage("\nEnter your PIN: "); // solicita o PIN
46 int pin = keypad.getInput(); // insere o PIN
47
48 // configura userAuthenticated como um valor bool retornado pelo banco de dados
49 userAuthenticated =
50 bankDatabase.authenticateUser(accountNumber, pin);
51
52 // verifica se a autenticação foi bem-sucedida
53 if (userAuthenticated)
54 {
55 currentAccountNumber = accountNumber; // salva a conta do usuário #
56 } // fim do if
57 else
58 screen.displayMessageLine(
59 "Invalid account number or PIN. Please try again.");
60 } // fim da função authenticateUser
61
62 // exibe o menu principal e realiza transações
63 void ATM::performTransactions()
64 {
65 // ponteiro local para armazenar a transação atualmente processada
66 Transaction *currentTransactionPtr;
67
68 bool userExited = false; // o usuário optou por não sair
69
70 // faz um loop enquanto o usuário não escolher a opção para sair do sistema
71 while (!userExited)
72 {
73 // mostra o menu principal e obtém a seleção de usuário
74 int mainMenuSelection = displayMainMenu();
75
76 // decide como prosseguir com base na seleção de menu feita pelo usuário
77 switch (mainMenuSelection)
78 {
79 // o usuário optou por realizar um entre três tipos de transação
80 case BALANCE_INQUIRY:
81 case WITHDRAWAL:
82 case DEPOSIT:
83 // inicializa como o novo objeto do tipo escolhido
84 currentTransactionPtr =

```

**Figura G.2** Definições de função-membro da classe ATM.

(continua)

```

85 createTransaction(mainMenuSelection);
86
87 currentTransactionPtr->execute(); // executa a transação
88
89 // libera o espaço para a Transaction dinamicamente alocada
90 delete currentTransactionPtr;
91
92 break;
93 case EXIT: // o usuário optou por terminar a sessão
94 screen.displayMessageLine("\nExiting the system..");
95 userExited = true; // essa sessão de ATM deve terminar
96 break;
97 default: // o usuário não inseriu um inteiro de 1 a 4
98 screen.displayMessageLine(
99 "\nYou did not enter a valid selection. Try again.");
100 break;
101 } // fim do switch
102 } // fim do while
103 } // fim da função performTransactions
104
105 // exibe o menu principal e retorna uma seleção de entrada
106 int ATM::displayMainMenu() const
107 {
108 screen.displayMessageLine("\nMain menu:");
109 screen.displayMessageLine("1 - View my balance");
110 screen.displayMessageLine("2 - Withdraw cash");
111 screen.displayMessageLine("3 - Deposit funds");
112 screen.displayMessageLine("4 - Exit\n");
113 screen.displayMessage("Enter a choice: ");
114 return keypad.getInput(); // retorna a seleção do usuário
115 } // fim da função displayMainMenu
116
117 // retorna o objeto da classe derivada Transaction especificada
118 Transaction *ATM::createTransaction(int type)
119 {
120 Transaction *tempPtr; // ponteiro Transaction temporário
121
122 // determina que tipo Transaction criar
123 switch (type)
124 {
125 case BALANCE_INQUIRY: // cria uma nova transação BalanceInquiry
126 tempPtr = new BalanceInquiry(
127 currentAccountNumber, screen, bankDatabase);
128 break;
129 case WITHDRAWAL: // cria uma nova transação Withdrawal
130 tempPtr = new Withdrawal(currentAccountNumber, screen,
131 bankDatabase, keypad, cashDispenser);
132 break;
133 case DEPOSIT: // cria uma nova transação Deposit
134 tempPtr = new Deposit(currentAccountNumber, screen,
135 bankDatabase, keypad, depositSlot);
136 break;
137 } // fim do switch
138
139 return tempPtr; // retorna o objeto recém-criado
140 } // fim da função createTransaction

```

Figura G.2 Definições de função-membro da classe ATM.

(continuação)

Dentro do loop infinito da função-membro `run`, as linhas 39–43 fazem com que o ATM repetidamente emita uma mensagem de boas-vindas e tente autenticar o usuário contanto que ele não tenha sido autenticado (isto é, `!userAuthenticated` é `true`). A linha 29 invoca a função-membro `displayMessageLine` da `screen` do ATM para exibir uma mensagem de boas-vindas. Como a função-membro `Screen displayMessage` projetada no estudo de caso, a função-membro `displayMessageLine` (declarada na linha 13 da Figura G.3 e definida nas linhas 20–23 da Figura G.4) exibe uma mensagem para o usuário, mas essa função-membro também gera saída de uma nova linha depois de exibir a mensagem. Adicionamos essa função-membro durante a implementação para fornecer aos clientes da classe `Screen` mais controle sobre o posicionamento das mensagens exibidas. A linha 30 da Figura G.2 invoca a função utilitária `private authenticateUser` da classe `ATM` (linhas 41–60) para tentar autenticar o usuário.

Recorremos ao documento de requisitos para determinar os passos necessários a fim de autenticar o usuário antes de permitir que as transações ocorram. A linha 43 da função-membro `authenticateUser` invoca a função-membro `displayMessage` da `screen` do ATM para solicitar que o usuário insira o número de uma conta. A linha 44 invoca a função `getInput` do `keypad` da classe `ATM` para obter a entrada do usuário e então armazena o valor inteiro inserido pelo usuário em uma variável local `accountNumber`. Em seguida, a função-membro `authenticateUser` solicita que o usuário insira um PIN (linha 45) e armazena o PIN inserido pelo usuário em uma variável local `pin` (linha 46). Em seguida, as linhas 49–50 tentam autenticar o usuário passando o `accountNumber` e o `pin` inseridos pelo usuário para a função-membro `authenticateUser` de `bankDatabase`. A classe `ATM` configura seu membro de dados `userAuthenticated` com o valor `bool` retornado por essa função — `userAuthenticated` torna-se `true` se a autenticação for bem-sucedida (isto é, `accountNumber` e `pin` correspondem aos de um `Account` existente em `bankDatabase`) e permanece `false`, caso contrário. Se `userAuthenticated` for `true`, a linha 55 salva o número de conta inserido pelo usuário (isto é, `accountNumber`) no membro de dados `ATM currentAccountNumber`. As outras funções-membro da classe `ATM` utilizam essa variável sempre que uma sessão de ATM exige acesso ao número da conta do usuário. Se `userAuthenticated` é `false`, as linhas 70–71 utilizam a função-membro `displayMessageLine` de `screen` para indicar que um número inválido de conta e/ou PIN foi inserido e que o usuário deve tentar novamente. Observe que configuramos `currentAccountNumber` somente depois de autenticar o número da conta do usuário e o PIN associado — se o banco de dados não puder autenticar o usuário, `currentAccountNumber` permanece 0.

Depois de a função-membro `run` tentar autenticar o usuário (linha 30), se `userAuthenticated` ainda for `false`, o loop `while` nas linhas 27–31 executa novamente. Se `userAuthenticated` for `true` agora, o loop terminará e o controle continua com a linha 33, que chama a função utilitária `performTransactions` da classe `ATM`.

A função-membro `performTransactions` (linhas 63–103) executa uma sessão ATM para um usuário autenticado. A linha 66 declara um ponteiro `Transaction` local, que apontamos para um objeto `BalanceInquiry`, `Withdrawal` ou `Deposit` que representa a transação ATM sendo atualmente processada. Observe que aqui utilizamos um ponteiro `Transaction` para permitir que tiremos proveito do polimorfismo. Também observe que utilizamos o nome de papel incluído no diagrama de classes da Figura 3.20 — `currentTransaction` — ao atribuir um nome a ponteiro. De acordo com nossa convenção de atribuição de nomes de ponteiro, acrescentamos ‘Ptr’ ao nome de papel para formar o nome da variável `currentTransactionPtr`. A linha 68 declara outra variável local — um `bool` chamado `userExited` que monitora se o usuário optou por sair. Essa variável controla um loop `while` (linhas 71–102) que permite ao usuário executar um número ilimitado de transações antes de optar por sair. Dentro desse loop, a linha 74 exibe o menu principal e obtém a seleção de menu do usuário chamando a função utilitária `displayMainMenu` de `ATM` (definida nas linhas 106–115). Essa função-membro exibe o menu principal invocando as funções-membro do `screen` do ATM e retorna uma seleção de menu obtida a partir do usuário pelo `keypad`

```

1 // Screen.h
2 // Definição da classe Screen. Representa a tela do ATM.
3 #ifndef SCREEN_H
4 #define SCREEN_H
5
6 #include <string>
7 using std::string;
8
9 class Screen
10 {
11 public:
12 void displayMessage(string) const; // gera saída de uma mensagem
13 void displayMessageLine(string) const; // gera saída da mensagem com nova linha
14 void displayDollarAmount(double) const; // gera saída de um valor em dólar
15 }; // fim da classe Screen
16
17 #endif // SCREEN_H

```

**Figura G.3** Definição da classe `Screen`.

```

1 // Screen.cpp
2 // Definições de função-membro para a classe Screen.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6 using std::fixed;
7
8 #include <iomanip>
9 using std::setprecision;
10
11 #include "Screen.h" // definição da classe Screen
12
13 // gera saída de uma mensagem sem uma nova linha
14 void Screen::displayMessage(string message) const
15 {
16 cout << message;
17 } // fim da função displayMessage
18
19 // gera saída de uma mensagem com uma nova linha
20 void Screen::displayMessageLine(string message) const
21 {
22 cout << message << endl;
23 } // fim da função displayMessageLine
24
25 // gera saída de um valor em dólar
26 void Screen::displayDollarAmount(double amount) const
27 {
28 cout << fixed << setprecision(2) << "$" << amount;
29 } // fim da função displayDollarAmount

```

**Figura G.4** Definições de função-membro da classe Screen.

do ATM. Observe que essa função-membro é `const` porque não modifica o conteúdo do objeto. A linha 74 armazena a seleção do usuário retornada por `displayMainMenu` na variável local `mainMenuSelection`.

Depois de obter uma seleção do menu principal, a função-membro `performTransactions` utiliza uma instrução `switch` (linhas 77–101) para responder à seleção apropriadamente. Se `mainMenuSelection` for igual a qualquer uma das três constantes de enumeração que representam os tipos de transação (isto é, se o usuário escolheu realizar uma transação), as linhas 84–85 chamam a função utilitária `createTransaction` (definida nas linhas 118–140) para retornar um ponteiro para um objeto recém-instanciado do tipo que corresponde à transação selecionada. O ponteiro retornado por `createTransaction` é atribuído ao ponteiro `currentTransactionPtr`. A linha 87 então utiliza `currentTransactionPtr` para invocar a função-membro `execute` do novo objeto a fim de executar a transação. Discutiremos a função-membro `Transaction::execute` e as três classes derivadas `Transaction` em breve. Por fim, quando o objeto `Transaction` de classe derivada não for mais necessário, a linha 90 libera a memória dinamicamente alocada para ele.

Observe que apontamos o ponteiro `Transaction::currentTransactionPtr` para objetos de uma das três classes derivadas `Transaction` para podermos executar as transações polimorficamente. Por exemplo, se o usuário escolher realizar uma consulta de saldos, `mainMenuSelection` será igual a `BALANCE_INQUIRY`, levando `createTransaction` a retornar um ponteiro para um objeto `BalanceInquiry`. Portanto, `currentTransactionPtr` aponta para um `BalanceInquiry`, e invocar `currentTransactionPtr->execute()` resulta em uma chamada a uma versão `execute` de `BalanceInquiry`.

A função-membro `createTransaction` (linhas 118–140) utiliza uma instrução `switch` (linhas 123–137) para instanciar um novo objeto da classe derivada `Transaction` do tipo indicado pelo parâmetro `type`. Lembre-se de que a função-membro `performTransactions` passa `mainMenuSelection` para essa função-membro apenas quando `mainMenuSelection` contiver um valor correspondente a um dos três tipos de transação. Portanto, `type` é igual a `BALANCE_INQUIRY`, `WITHDRAWAL` ou `DEPOSIT`. Cada case na instrução `switch` aponta o ponteiro temporário `tempPtr` para um objeto criado recentemente da classe derivada `Transaction` apropriada. Observe que cada construtor contém uma lista única de parâmetros, baseada nos dados específicos necessários para inicializar o objeto da classe derivada. Uma `BalanceInquiry` exige apenas o número de conta do usuário atual e referências à `bankDatabase` e `screen` da classe `ATM`. Além desses parâmetros, um `Withdrawal` requer referências ao `keypad` e ao `cashDispenser` da classe `ATM`, e um `Deposit` requer referências ao `keypad` e ao `depositSlot` da classe `ATM`. Observe que, como você logo verá, os construtores `BalanceInquiry`, `Withdrawal` e `Deposit` especificam parâmetros de referência para receber os objetos que representam as partes requeridas do ATM. Portanto, quando a



função-membro `createTransaction` passa objetos no ATM (por exemplo, `screen` e `keypad`) para o inicializador de cada objeto da classe derivada `Transaction` recentemente criado, na realidade, o novo objeto recebe *referências* aos objetos compostos do ATM. Discutimos as classes de transação em mais detalhes nas seções G.9–G.12.

Depois de executar uma transação (linha 87 em `performTransactions`), `userExited` permanece `false` e o loop `while` nas linhas 71–102 é repetido, retornando o usuário ao menu principal. Entretanto, se um usuário não realizar uma transação e, em vez disso, selecionar a opção de saída no menu principal, a linha 95 configura `userExited` como `true`, fazendo com que a condição do loop `while` (`!userExited`) se torne `false`. Esse `while` é a instrução final da função-membro `performTransactions`, portanto o controle retorna à função chamadora `run`. Se o usuário inserir uma seleção inválida no menu principal (isto é, não um inteiro entre 1 e 4), as linhas 98–99 exibem uma mensagem de erro apropriada, `userExited` permanece `false` e o usuário retorna ao menu principal para tentar novamente.

Quando `performTransactions` retorna o controle à função-membro `run`, o usuário escolheu sair do sistema, portanto as linhas 34–35 reinicializam os membros de dados `userAuthenticated` e `currentAccountNumber` do ATM para se preparar para o próximo usuário ATM. A linha 36 exibe uma mensagem de adeus antes de o ATM recomeçar e dar boas-vindas ao próximo usuário.

## G.3 Classe Screen

A classe `Screen` (figuras G.3–G.4) representa a tela do ATM e encapsula todos os aspectos da exibição da saída para o usuário. A classe `Screen` simula a tela de um ATM real com um monitor de computador e gera saída de mensagens de texto utilizando `cout` e o operador de inserção de fluxo (`<<`). Nesse estudo de caso, projetamos a classe `Screen` com uma operação — `displayMessage`. Para maior flexibilidade na exibição de mensagens a `Screen`, declaramos agora três funções-membro `Screen` — `displayMessage`, `displayMessageLine` e `displayDollarAmount`. Os protótipos para essas funções-membro aparecem nas linhas 12–14 da Figura G.3.

### Definições de função-membro da classe `Screen`

A Figura G.4 contém as definições de função-membro da classe `Screen`. A linha 11 inclui (`#include`) a definição da classe `Screen`. A função-membro `displayMessage` (linhas 14–17) aceita uma `string` como um argumento e a imprime para o console utilizando `cout` e o operador de inserção de fluxo (`<<`). O cursor permanece na mesma linha, tornando essa função-membro apropriada para exibir solicitações para o usuário. A função-membro `displayMessageLine` (linhas 20–23) também imprime uma `string`, mas gera saída de uma nova linha para mover o cursor para a próxima linha. Por fim, a função-membro `displayDollarAmount` (linhas 26–29) gera saída de um valor em dólar adequadamente formatado (por exemplo, \$123.45). A linha 28 utiliza os manipuladores de fluxo `fixed` e `setprecision` para gerar saída de um valor formatado com duas casas decimais. Veja o Capítulo 15, “Entrada/saída de fluxo”, para obter informações adicionais sobre a formatação de saída.

## G.4 Classe Keypad

A classe `Keypad` (figuras G.5–G.6) representa o teclado do ATM e é responsável por receber toda a entrada do usuário. Lembre-se de que estamos simulando esse hardware, portanto utilizamos o teclado do computador para simular o teclado numérico. O teclado de um computador contém muitas teclas não encontradas no teclado numérico do ATM. Entretanto, supomos que o usuário pressione somente as teclas presentes no teclado de um computador que também aparecem no teclado numérico — as teclas numeradas de 0–9 e a tecla `Enter`. A linha 9 da Figura G.5 contém o protótipo de função para uma função-membro `getInput` da classe `Keypad`. Essa função-membro é declarada `const` porque não altera o objeto.

### Definição de função-membro da classe `Keypad`

No arquivo de implementação `Keypad` (Figura G.6), a função-membro `getInput` (definida nas linhas 9–14) utiliza o operador de fluxo de entrada-padrão `cin` e o operador de extração de fluxo (`>>`) para obter a entrada do usuário. A linha 11 declara uma variável local

```

1 // Keypad.h
2 // Definição da classe Keypad. Representa o teclado do ATM.
3 #ifndef KEYPAD_H
4 #define KEYPAD_H
5
6 class Keypad
7 {
8 public:
9 int getInput() const; // retorna um valor de inteiro inserido pelo usuário
10 }; // fim da classe Keypad
11
12 #endif // KEYPAD_H

```

**Figura G.5** Definição da classe `Keypad`.



```

1 // Keypad.cpp
2 // Definição da função-membro da classe Keypad (o teclado do ATM).
3 #include <iostream>
4 using std::cin;
5
6 #include "Keypad.h" // definição da classe Keypad
7
8 // retorna um valor de inteiro inserido pelo usuário
9 int Keypad::getInput() const
10 {
11 int input; // variável para armazenar a entrada
12 cin >> input; // supomos que o usuário insere um inteiro
13 return input; // retorna o valor inserido pelo usuário
14 } // fim da função getInput

```

**Figura G.6** Definição de função-membro da classe Keypad.

para armazenar a entrada do usuário. A linha 12 lê a entrada na variável local `input` e, então, a linha 13 retorna esse valor. Lembre-se de que `getInput` obtém todas as entradas utilizadas pelo ATM. A função-membro `getInput` do `Keypad` simplesmente retorna a entrada de inteiro pelo usuário. Se um cliente da classe `Keypad` exigir uma entrada que deve satisfazer alguns critérios em particular (isto é, um número que corresponde a uma opção válida no menu), o cliente deverá realizar a verificação de erros apropriada. [Nota: Utilizar o operador de fluxo de entrada-padrão `cin` e o operador de extração de fluxo (`>>`) permite que a entrada de tipo não-inteiro seja lida a partir do usuário. Entretanto, como o teclado do ATM permite apenas a entrada de inteiro, pressupomos que o usuário insere um inteiro e não tenta corrigir os problemas causados pela entrada de tipo não-inteiro.]

## G.5 Classe CashDispenser

A classe `CashDispenser` (figuras G.7–G.8) representa o dispensador de cédulas do ATM. A definição de classe (Figura G.7) contém o protótipo de função de um construtor-padrão (linha 9). A classe `CashDispenser` declara duas funções-membro `public` adicionais — `dispenseCash` (linha 12) e `isSufficientCashAvailable` (linha 15). A classe confia no fato de que um cliente (isto é, `Withdrawal`) chama `dispenseCash` somente depois de estabelecer que há cédulas suficientes disponíveis chamando `isSufficientCashAvailable`.

```

1 // CashDispenser.h
2 // Definição da classe CashDispenser. Representa o ATM para o dispensador de cédulas.
3 #ifndef CASH_DISPENSER_H
4 #define CASH_DISPENSER_H
5
6 class CashDispenser
7 {
8 public:
9 CashDispenser(); // o construtor inicializa contagem de conta para 500
10
11 // simula a entrega da quantia especificada de cédulas
12 void dispenseCash(int);
13
14 // indica se o dispensador de cédulas pode entregar a quantia desejada
15 bool isSufficientCashAvailable(int) const;
16 private:
17 const static int INITIAL_COUNT = 500;
18 int count; // número de cédulas de $20 remanescente
19 }; // fim da classe CashDispenser
20
21 #endif // CASH_DISPENSER_H

```

**Figura G.7** Definição da classe CashDispenser.

Portanto, `dispenseCash` simplesmente simula o ato de entregar a quantia solicitada sem verificar se há cédulas suficientes disponíveis. A linha 17 declara a constante `private INITIAL_COUNT`, que indica a contagem inicial de cédulas no dispensador de cédulas quando o ATM é inicializado (isto é, 500). A linha 18 implementa o atributo `count` (modelado na Figura 13.29), que monitora o número de cédulas que permanece no `CashDispenser` em um dado momento.

### Definições de função-membro da classe *CashDispenser*

A Figura G.8 contém as definições das funções-membro da classe `CashDispenser`. O construtor (linhas 6–9) configura `count` como a contagem inicial (isto é, 500). A função-membro `dispenseCash` (linhas 13–17) simula a liberação do dinheiro. Se nosso sistema estivesse acoplado ao dispensador de cédulas de um hardware real, essa função-membro interagiria com o dispositivo de hardware para fisicamente entregar as cédulas. Nossa versão simulada da função-membro simplesmente subtrairia a `count` de cédulas remanescentes de acordo com o número requerido para entregar a `amount` especificada (linha 16). Observe que a linha 15 calcula o número de cédulas de \$20 necessárias para liberar a `amount` especificada. O ATM permite que o usuário escolha somente quantias de retirada que são múltiplos de \$20, assim dividimos `amount` por 20 para obter o número de `billsRequired`. Observe também que é responsabilidade do cliente da classe (isto é, `Withdrawal`) informar o usuário de que cédulas foram entregues — `CashDispenser` não pode interagir diretamente com `Screen`.

A função-membro `isSufficientCashAvailable` (linhas 20–28) tem um parâmetro `amount` que especifica a quantia de dinheiro em questão. As linhas 24–26 retornam `true` se a `count` de `CashDispenser` for maior ou igual a `billsRequired` (isto é, há cédulas suficientes disponíveis) e `false` caso contrário (isto é, não há notas suficientes). Por exemplo, se um usuário deseja sacar \$80 (isto é, `billsRequired` é 4), mas só há três cédulas (isto é, `count` é 3), a função-membro retorna `false`.

## G.6 Classe DepositSlot

A classe `DepositSlot` (figuras G.9–G.10) representa a abertura de depósito do ATM. Como a versão da classe `CashDispenser` apresentada aqui, essa versão da classe `DepositSlot` meramente simula as funcionalidades de uma abertura de depósito de hardware real. `DepositSlot` não tem nenhum membro de dados e tem apenas uma função-membro — `isEnvelopeReceived` (declarada na linha 9 da Figura G.9 e definida nas linhas 7–10 da Figura G.10) — isso indica se um envelope de depósito foi ou não recebido.

```

1 // CashDispenser.cpp
2 // Definições de função-membro da classe CashDispenser.
3 #include "CashDispenser.h" // definição da classe CashDispenser
4
5 // o construtor-padrão CashDispenser inicializa a contagem como padrão
6 CashDispenser::CashDispenser()
7 {
8 count = INITIAL_COUNT; // configura atributo count como o padrão
9 } // fim do construtor-padrão CashDispenser
10
11 // simula a liberação da quantia em dinheiro especificada; supõe que há dinheiro suficiente
12 // disponível (chamada anterior para isSufficientCashAvailable retornou true)
13 void CashDispenser::dispenseCash(int amount)
14 {
15 int billsRequired = amount / 20; // número de cédulas de $20 requerido
16 count -= billsRequired; // atualiza a contagem das cédulas
17 } // fim da função dispenseCash
18
19 // indica se o dispensador de cédulas pode entregar a quantia desejada
20 bool CashDispenser::isSufficientCashAvailable(int amount) const
21 {
22 int billsRequired = amount / 20; // número de cédulas de $20 requerido
23
24 if (count >= billsRequired)
25 return true; // cédulas suficientes estão disponíveis
26 else
27 return false; // não há cédulas suficientes disponíveis
28 } // fim da função isSufficientCashAvailable

```

**Figura G.8** Definições de função-membro da classe `CashDispenser`.

```

1 // DepositSlot.h
2 // Definição da classe DepositSlot. Representa a abertura de depósito do ATM.
3 #ifndef DEPOSIT_SLOT_H
4 #define DEPOSIT_SLOT_H
5
6 class DepositSlot
7 {
8 public:
9 bool isEnvelopeReceived() const; // informa se o envelope foi recebido
10 }; // fim da classe DepositSlot
11
12 #endif // DEPOSIT_SLOT_H

```

**Figura G.9** Definição da classe `DepositSlot`.

```

1 // DepositSlot.cpp
2 // Definição de função-membro da classe DepositSlot.
3 #include "DepositSlot.h" // definição da classe DepositSlot
4
5 // indica se o envelope foi recebido (sempre retorna true,
6 // porque esta é apenas uma simulação do software de uma abertura para depósito real)
7 bool DepositSlot::isEnvelopeReceived() const
8 {
9 return true; // o envelope de depósito foi recebido
10 } // fim da função isEnvelopeReceived

```

**Figura G.10** Definição de função-membro da classe `DepositSlot`.

Lembre-se de que, no documento de requisitos, o ATM permite que o usuário insira um envelope dentro de no máximo dois minutos. A versão atual da função-membro `isEnvelopeReceived` simplesmente retorna `true` imediatamente (linha 9 da Figura G.10), porque esta é apenas uma simulação de software e pressupomos que o usuário inseriu um envelope dentro do período de tempo requerido. Se um hardware real de abertura para depósito estivesse conectado ao nosso sistema, a função-membro `isEnvelopeReceived` poderia ser implementada para esperar no máximo dois minutos a fim de receber um sinal do hardware da abertura para depósito indicando que o usuário de fato inseriu um envelope de depósito. Se `isEnvelopeReceived` recebesse esse sinal dentro de dois minutos, a função-membro retornaria `true`. Se passassem dois minutos e a função-membro ainda não tivesse recebido um sinal, então a função-membro retornaria `false`.

## G.7 Classe `Account`

A classe `Account` (figuras G.11–G.12) representa uma conta bancária. As linhas 9–15 na definição de classe (Figura G.11) contêm protótipos de função para o construtor da classe e seis funções-membro, que discutiremos em breve. Cada `Account` tem quatro atributos (modelados na Figura 13.29) — `accountNumber`, `pin`, `availableBalance` e `totalBalance`. As linhas 17–20 implementam esses atributos como membros de dados `private`. O membro de dados `availableBalance` representa a quantia de fundos disponíveis para a retirada. O membro de dados `totalBalance` representa a quantia de fundos disponível, mais a quantia de fundos depositados ainda aguardando confirmação ou compensação.

### *Definições de função-membro da classe `Account`*

A Figura G.12 apresenta as definições das funções-membro da classe `Account`. O construtor da classe (linhas 6–14) aceita um número de conta, o PIN estabelecido para a conta, o saldo disponível inicial e o saldo total inicial como argumentos. As linhas 8–11 atribuem esses valores aos membros de dados da classe utilizando inicializadores de membro.

A função-membro `validatePIN` (linhas 17–23) determina se um PIN especificado pelo usuário (isto é, o parâmetro `userPIN`) corresponde ao PIN associado com a conta (isto é, o membro de dados `pin`). Lembre-se de que modelamos o parâmetro `userPIN` dessa função-membro no diagrama da classe UML da Figura 6.37. Se os dois PINs corresponderem, a função-membro retornará `true` (linha 20); caso contrário, retornará `false` (linha 22).

```

1 // Account.h
2 // Definição da classe Account. Representa uma conta bancária.
3 #ifndef ACCOUNT_H
4 #define ACCOUNT_H
5
6 class Account
7 {
8 public:
9 Account(int, int, double, double); // o construtor configura atributos
10 bool validatePIN(int) const; // o PIN especificado pelo usuário é correto?
11 double getAvailableBalance() const; // retorna o saldo disponível
12 double getTotalBalance() const; // retorna o saldo total
13 void credit(double); // adiciona um valor ao de Account
14 void debit(double); // subtrai uma quantia do saldo de Account
15 int getAccountNumber() const; // retorna o número da conta
16 private:
17 int accountNumber; // número da conta
18 int pin; // PIN para autenticação
19 double availableBalance; // fundos disponíveis para retirada
20 double totalBalance; // fundos disponíveis + fundos esperando compensação
21 }; // fim da classe Account
22
23 #endif // ACCOUNT_H

```

**Figura G.11** Definição da classe Account.

```

1 // Account.cpp
2 // Definições de função-membro para a classe Account.
3 #include "Account.h" // definição da classe Account
4
5 // o construtor Account inicializa os atributos
6 Account::Account(int theAccountNumber, int thePIN,
7 double theAvailableBalance, double theTotalBalance)
8 : accountNumber(theAccountNumber),
9 pin(thePIN),
10 availableBalance(theAvailableBalance),
11 totalBalance(theTotalBalance)
12 {
13 // corpo vazio
14 } // fim do construtor Account
15
16 // determina se um PIN especificado pelo usuário corresponde ao PIN em Account
17 bool Account::validatePIN(int userPIN) const
18 {
19 if (userPIN == pin)
20 return true;
21 else
22 return false;
23 } // fim da função validatePIN
24
25 // retorna o saldo disponível
26 double Account::getAvailableBalance() const
27 {
28 return availableBalance;

```

**Figura G.12** Definições de função-membro da classe Account.

(continua)

```

29 } // fim da função getAvailableBalance
30
31 // retorna o saldo total
32 double Account::getTotalBalance() const
33 {
34 return totalBalance;
35 } // fim da função getTotalBalance
36
37 // credita uma quantia à conta
38 void Account::credit(double amount)
39 {
40 totalBalance += amount; // adiciona ao saldo total
41 } // fim da função credit
42
43 // debita uma quantia da conta
44 void Account::debit(double amount)
45 {
46 availableBalance -= amount; // subtrai do saldo disponível
47 totalBalance -= amount; // subtrai do saldo total
48 } // fim da função debit
49
50 // retorna o número da conta
51 int Account::getAccountNumber() const
52 {
53 return accountNumber;
54 } // fim da função getAccountNumber

```

Figura G.12 Definições de função-membro da classe Account.

(continuação)

As funções-membro `getAvailableBalance` (linhas 26–29) e `getTotalBalance` (linhas 32–35) são as funções *get* que retornam os valores de membros de dados `availableBalance` e `totalBalance`, respectivamente.

A função-membro `credit` (linhas 38–41) adiciona uma quantia de dinheiro (isto é, o parâmetro `amount`) a uma `Account` como parte de uma transação de depósito. Observe que essa função-membro adiciona a `amount` somente ao membro de dados `totalBalance` (linha 40). O dinheiro creditado em uma conta durante um depósito não é disponibilizado imediatamente, portanto modificamos apenas o saldo total. Supomos que o banco atualize o saldo disponível apropriadamente em um momento posterior. Nossa implementação da classe `Account` inclui apenas as funções-membro necessárias para executar as transações ATM. Portanto, omitimos as funções-membro que algum outro sistema bancário invocaria para adicionar o membro de dados `availableBalance` (para confirmar um depósito) ou subtrair do membro de dados `totalBalance` (para rejeitar um depósito).

A função-membro `debit` (linhas 44–48) subtrai uma quantia de dinheiro (isto é, o parâmetro `amount`) de uma `Account` como parte de uma transação de retirada. Essa função-membro subtrai a `amount` de ambos os membros de dados `availableBalance` (linha 46) e `totalBalance` (linha 47), porque uma retirada afeta ambas as medidas de um saldo da conta.

A função-membro `getAccountNumber` (linhas 51–54) fornece acesso ao `accountNumber` de uma `Account`. Incluímos essa função-membro na nossa implementação para que um cliente da classe (isto é, `BankDatabase`) possa identificar uma `Account` particular. Por exemplo, `BankDatabase` contém muitos objetos `Account` e pode invocar essa função-membro em cada um dos seus objetos `Account` para localizar aquele com um número específico de conta.

## G.8 Classe BankDatabase

A classe `BankDatabase` (figuras G.13–G.14) modela o banco de dados do banco com o qual o ATM interage para acessar e modificar informações da conta de um usuário. A definição de classe (Figura G.13) declara protótipos de função para o construtor da classe e várias funções-membro. Discutiremos esses arquivos em breve. A definição de classe também declara os membros de dados do `BankDatabase`. Determinamos um membro de dados para a classe `BankDatabase` com base em seu relacionamento de composição com a classe `Account`. Lembre-se de que, na Figura 13.28, um `BankDatabase` é composto de zero ou mais objetos da classe `Account`. A linha 24 da Figura G.13 implementa o membro de dados `accounts` — um vector de objetos `Account` — para implementar esse relacionamento de composição. As linhas 6–7 permitem utilizar vector nesse arquivo. A linha 27 contém o protótipo de função para uma função utilitária `private` `getAccount` que permite às funções-membro da classe obter um ponteiro para uma `Account` específica no `accounts` vector.

```

1 // BankDatabase.h
2 // Definição da classe BankDatabase. Representa o banco de dados do banco.
3 #ifndef BANK_DATABASE_H
4 #define BANK_DATABASE_H
5
6 #include <vector> // a classe utiliza vector para armazenar objetos Account
7 using std::vector;
8
9 #include "Account.h" // definição da classe Account
10
11 class BankDatabase
12 {
13 public:
14 BankDatabase(); // o construtor inicializa as contas
15
16 // determina se o número de conta e o PIN correspondem aos de uma Account
17 bool authenticateUser(int, int); // retorna true se a Account for autêntica
18
19 double getAvailableBalance(int); // obtém um saldo disponível
20 double getTotalBalance(int); // obtém o saldo total de uma Account
21 void credit(int, double); // adiciona o valor ao saldo de Account
22 void debit(int, double); // subtrai o valor do saldo de Account
23 private:
24 vector< Account > accounts; // vector das Accounts do banco
25
26 // função utilitária private
27 Account * getAccount(int); // obtém ponteiro para o objeto Account
28 }; // fim da classe BankDatabase
29
30 #endif // BANK_DATABASE_H

```

**Figura G.13** Definição da classe BankDatabase.

### Definições de função-membro da classe *BankDatabase*

A Figura G.14 contém as definições de função-membro da classe *BankDatabase*. Implementamos a classe com um construtor padrão (linhas 6–15) que adiciona objetos *Account* ao membro de dados *accounts*. Por testar o sistema, criamos dois novos objetos *Account* com dados de teste (linhas 9–10), então os adicionamos ao fim do vector (linhas 13–14). Observe que o construtor *Account* tem quatro parâmetros — o número de conta, o PIN atribuído à conta, o saldo inicial disponível e o saldo inicial total.

Lembre-se de que a classe *BankDatabase* serve como um intermediário entre a classe *ATM* e os objetos *Account* reais que contêm informações de conta dos usuários. Portanto, as funções-membro da classe *BankDatabase* não fazem nada mais que invocar as funções-membro correspondentes do objeto *Account* que pertence ao usuário *ATM* atual.

Incluimos a função utilitária private *getAccount* (linhas 18–29) para permitir a *BankDatabase* obter um ponteiro para um objeto *Account* particular dentro de vector *accounts*. Para localizar a *Account* do usuário, *BankDatabase* compara o valor retornado pela função-membro *getAccountNumber* de cada elemento de *accounts* com um número especificado de conta até encontrar uma correspondência. As linhas 21–26 percorrem o *accounts* vector. Se o número da conta da *Account* atual (isto é, *accounts[ i ]*) for igual ao valor do parâmetro *accountNumber*, a função-membro retorna imediatamente o endereço da *Account* atual (isto é, um ponteiro para a *Account* atual). Se nenhuma conta tiver o número de conta dado, a linha 28 então retornará *NULL*. Observe que essa função-membro deve retornar um ponteiro, em oposição a uma referência, porque há a possibilidade de o valor de retorno ser *NULL* — uma referência não pode ser *NULL*, mas um ponteiro pode.

Observe que a função *vector::size* (invocada na condição de continuação do loop na linha 21) retorna o número de elementos em um vector como um valor do tipo *size\_t* (que, normalmente, é *unsigned int*). Como resultado, declaramos a variável de controle *i* como sendo do tipo *size\_t*, também. Em alguns compiladores, declarar *i* como um *int* faria com que o compilador emitisse uma mensagem de advertência, porque a condição de continuação do loop compararia um valor *signed* (isto é, um *int*) e um valor *unsigned* (isto é, um valor do tipo *size\_t*).

A função-membro *authenticateUser* (linhas 33–44) comprova a identidade de um usuário *ATM*. Essa função-membro recebe um número e um PIN de conta especificados pelo usuário como argumentos e indica se eles correspondem ao número da conta e ao PIN de uma *Account* no banco de dados. A linha 37 chama a função utilitária *getAccount*, que retorna um ponteiro para uma *Account* com

```

1 // BankDatabase.cpp
2 // Definições da função-membro da classe BankDatabase.
3 #include "BankDatabase.h" // definição da classe BankDatabase
4
5 // construtor-padrão BankDatabase inicializa contas
6 BankDatabase::BankDatabase()
7 {
8 // cria dois objetos Account para testar
9 Account account1(12345, 54321, 1000.0, 1200.0);
10 Account account2(98765, 56789, 200.0, 200.0);
11
12 // adiciona os objetos Account ao vector accounts
13 accounts.push_back(account1); // adiciona account1 ao fim de vector
14 accounts.push_back(account2); // adiciona account2 ao fim de vector
15 } // fim do construtor-padrão BankDatabase
16
17 // recupera o objeto Account que contém o número de conta especificado
18 Account * BankDatabase::getAccount(int accountNumber)
19 {
20 // faz um loop pelas contas procurando uma correspondência com o número de conta
21 for (size_t i = 0; i < accounts.size(); i++)
22 {
23 // retorna a conta atual se uma correspondência for localizada
24 if (accounts[i].getAccountNumber() == accountNumber)
25 return &accounts[i];
26 } // fim do for
27
28 return NULL; // se nenhuma correspondência com uma conta foi encontrada, retorna NULL
29 } // fim da função getAccount
30
31 // determina se o número da conta e o PIN especificados pelo usuário correspondem
32 // àqueles de uma conta no banco de dados
33 bool BankDatabase::authenticateUser(int userAccountNumber,
34 int userPIN)
35 {
36 // tenta recuperar a conta com o número da conta
37 Account * const userAccountPtr = getAccount(userAccountNumber);
38
39 // se a conta existir, retorna o resultado da função validatePIN de Account
40 if (userAccountPtr != NULL)
41 return userAccountPtr->validatePIN(userPIN);
42 else
43 return false; // o número de conta não foi localizado, portanto retorna false
44 } // fim da função authenticateUser
45
46 // retorna o saldo disponível de Account com o número da conta especificado
47 double BankDatabase::getAvailableBalance(int userAccountNumber)
48 {
49 Account * const userAccountPtr = getAccount(userAccountNumber);
50 return userAccountPtr->getAvailableBalance();
51 } // fim da função getAvailableBalance
52
53 // retorna o saldo total de Account com o número da conta especificado
54 double BankDatabase::getTotalBalance(int userAccountNumber)
55 {
56 Account * const userAccountPtr = getAccount(userAccountNumber);

```

Figura G.14 Definições de função-membro da classe BankDatabase.

(continua)



```

57 return userAccountPtr->getTotalBalance();
58 } // fim da função getTotalBalance
59
60 // credita uma quantia a Account com o número da conta especificado
61 void BankDatabase::credit(int userAccountNumber, double amount)
62 {
63 Account * const userAccountPtr = getAccount(userAccountNumber);
64 userAccountPtr->credit(amount);
65 } // fim da função credit
66
67 // debita uma quantia da Account com o número da conta especificado
68 void BankDatabase::debit(int userAccountNumber, double amount)
69 {
70 Account * const userAccountPtr = getAccount(userAccountNumber);
71 userAccountPtr->debit(amount);
72 } // fim da função debit

```

**Figura G.14** Definições de função-membro da classe BankDatabase.

(continuação)

userAccountNumber como seu número de conta ou NULL para indicar que userAccountNumber é inválido. Declaramos userAccountPtr como um ponteiro const porque, uma vez que a função-membro aponta esse ponteiro para o usuário Account, o ponteiro não deve mudar. Se getAccount retorna um ponteiro para um objeto Account, a linha 41 retorna o valor bool retornado pela função-membro validatePIN desse objeto. Observe que a função-membro authenticateUser do BankDatabase não realiza a comparação de PIN em si — em vez disso, ela encaminha userPIN para a função-membro validatePIN do objeto Account para fazer isso. O valor retornado pela função-membro validatePIN de Account indica se o PIN especificado pelo usuário corresponde ao PIN da Account do usuário, portanto a função-membro authenticateUser simplesmente retorna esse valor ao cliente da classe (isto é, ATM).

A função-membro BankDatabase confia na classe ATM para invocar a função-membro authenticateUser e receber um valor de retorno true antes de permitir que o usuário realize as transações. BankDatabase também confia no fato de que cada objeto Transaction criado pela ATM contém o número de conta válida do usuário autenticado e que esse é o número de conta passado para as funções-membro BankDatabase remanescentes como o argumento userAccountNumber. Portanto, as funções-membro getAvailableBalance (linhas 47–51), getTotalBalance (linhas 54–58), credit (linhas 61–65) e debit (linhas 68–72) simplesmente recuperam um ponteiro para o objeto Account do usuário com a função utilitária getAccount e, então, utilizam esse ponteiro para invocar a função-membro Account apropriada no objeto Account do usuário. Sabemos que as chamadas a getAccount dentro dessas funções-membro nunca retornarão NULL, porque userAccountNumber deve referenciar uma Account existente. Observe que getAvailableBalance e getTotalBalance retornam os valores retornados pelas funções-membro Account correspondentes. Observe também que credit e debit simplesmente redirecionam o parâmetro amount para as funções-membro Account que eles invocam.

## G.9 Classe Transaction

A classe Transaction (figuras G.15–G.16) é uma classe básica abstrata que representa a noção de uma transação ATM. Ela contém os recursos comuns das classes derivadas BalanceInquiry, Withdrawal e Deposit. A Figura G.15 se expande no arquivo de cabeçalho Transaction desenvolvido inicialmente na Seção 13.10. As linhas 13, 17–19 e 22 contêm protótipos de função para o construtor e quatro funções-membro da classe, que discutiremos em breve. A linha 15 define um destrutor virtual com um corpo vazio — isso faz todos os destrutores de classe derivada virtual (mesmo aqueles definidos implicitamente pelo compilador) e assegura que os objetos de classe derivada alocados dinamicamente sejam destruídos de modo adequado quando forem excluídos via ponteiro de classe básica. As linhas 24–26 declaram os membros de dados private da classe. Lembre-se, no diagrama de classes da Figura 13.29, de que a classe Transaction contém um atributo accountNumber (implementado na linha 24) que indica a conta envolvida na Transaction. Derivamos os membros de dados screen (linha 25) e bankDatabase (linha 26) das associações da classe Transaction modeladas na Figura 13.28 — todas as transações exigem acesso à tela do ATM e ao banco de dados do banco, portanto incluímos referências a uma Screen e a um BankDatabase como membros de dados da classe Transaction. Como você logo verá, o construtor de Transaction inicializa essas referências. Observe que as declarações forward nas linhas 6–7 significam que o arquivo de cabeçalho contém referências a objetos das classes Screen e BankDatabase, mas que as definições dessas classes residem fora do arquivo de cabeçalho.

A classe Transaction tem um construtor (declarado na linha 13 da Figura G.15 e definido nas linhas 8–15 da Figura G.16) que aceita o número da conta do usuário atual e referências à tela do ATM e ao banco de dados da instituição financeira como argumentos. Como Transaction é uma classe abstrata, esse construtor nunca será chamado diretamente para instanciar os objetos de Transaction. Em vez disso, os construtores das classes derivadas Transaction utilizarão a sintaxe inicializadora da classe básica para invocar esse construtor.

```

1 // Transaction.h
2 // Definição da classe básica abstrata Transaction.
3 #ifndef TRANSACTION_H
4 #define TRANSACTION_H
5
6 class Screen; // declaração antecipada da classe Screen
7 class BankDatabase; // declaração antecipada da classe BankDatabase
8
9 class Transaction
10 {
11 public:
12 // o construtor inicializa recursos comuns de todas as Transactions
13 Transaction(int, Screen &, BankDatabase &);
14
15 virtual ~Transaction() { } // destrutor virtual com corpo vazio
16
17 int getAccountNumber() const; // retorna o número da conta
18 Screen &getScreen() const; // retorna a referência à tela
19 BankDatabase &getBankDatabase() const; // retorna referência ao banco de dados
20
21 // função virtual pura para realizar a transação
22 virtual void execute() = 0; // sobrescrita em classes derivadas
23 private:
24 int accountNumber; // indica conta envolvida
25 Screen &screen; // referência à tela ATM
26 BankDatabase &bankDatabase; // referência ao banco de dados de informações de conta
27 }; // fim da classe Transaction
28
29 #endif // TRANSACTION_H

```

**Figura G.15** Definição da classe Transaction.

A classe Transaction tem três funções *get* public — *getAccountNumber* (declarada na linha 17 da Figura G.15 e definida nas linhas 18–21 da Figura G.16), *getScreen* (declarada na linha 18 da Figura G.15 e definida nas linhas 24–27 da Figura G.16) e *getBankDatabase* (declarada na linha 19 da Figura G.15 e definida nas linhas 30–33 da Figura G.16). As classes derivadas Transaction herdam essas funções-membro de Transaction e as utilizam para ganhar acesso aos membros de dados private da classe Transaction.

A classe Transaction também declara uma função pura virtual *execute* (linha 22 da Figura G.15). Não faz sentido fornecer uma implementação para essa função-membro, pois uma transação genérica não pode ser executada. Portanto, declaramos essa função-membro como uma função virtual pura e forçamos cada classe derivada Transaction a fornecer sua própria implementação concreta que executa esse tipo particular de transação.

## G.10 Classe BalanceInquiry

A classe BalanceInquiry (figuras G.17–G.18) deriva da classe básica abstrata Transaction e representa uma transação ATM de consulta de saldo. BalanceInquiry não tem nenhum membro de dados próprio, mas herda os membros de dados Transaction *accountNumber*, *screen* e *bankDatabase*, que são acessíveis pelas funções *get* public de Transaction. Observe que a linha 6 inclui (*#include*) a definição da classe básica Transaction. O construtor BalanceInquiry (declarado na linha 11 da Figura G.17 e definido nas linhas 8–13 da Figura G.18) aceita argumentos correspondentes para os membros de dados Transaction e simplesmente os encaminha para o construtor Transaction, utilizando a sintaxe inicializadora da classe básica (linha 10 da Figura G.18). A linha 12 da Figura G.17 contém o protótipo de função para a função-membro *execute*, que é requerida para indicar a intenção de sobrescrever a função virtual pura da classe básica de mesmo nome.

A classe BalanceInquiry sobrescreve a função virtual pura *execute* de Transaction para fornecer uma implementação concreta (linhas 16–37 da Figura G.18) que realiza os passos envolvidos em uma consulta de saldo. As linhas 19–20 obtêm referências ao banco de dados da instituição financeira e à tela do ATM invocando funções-membro herdadas da classe básica Transaction. As linhas 23–24 recuperam o saldo disponível da conta envolvido invocando a função-membro *getAvailableBalance* de *bankDatabase*. Observe que a linha 24 utiliza a função-membro herdada *getAccountNumber* para obter o número da conta do usuário atual, que ela então passa para *getAvailableBalance*. As linhas 27–28 recuperam o saldo total da conta do usuário atual. As linhas 31–36 exibem as informações sobre

```

1 // Transaction.cpp
2 // Definições de função-membro da classe Transaction.
3 #include "Transaction.h" // definição da classe Transaction
4 #include "Screen.h" // definição da classe Screen
5 #include "BankDatabase.h" // definição da classe BankDatabase
6
7 // o construtor inicializa recursos comuns de todas as Transactions
8 Transaction::Transaction(int userAccountNumber, Screen &atmScreen,
9 BankDatabase &atmBankDatabase)
10 : accountNumber(userAccountNumber),
11 screen(atmScreen),
12 bankDatabase(atmBankDatabase)
13 {
14 // corpo vazio
15 } // fim do construtor de Transaction
16
17 // retorna o número da conta
18 int Transaction::getAccountNumber() const
19 {
20 return accountNumber;
21 } // fim da função getAccountNumber
22
23 // retorna a referência à tela
24 Screen &Transaction::getScreen() const
25 {
26 return screen;
27 } // fim da função getScreen
28
29 // retorna a referência ao banco de dados de instituição financeira
30 BankDatabase &Transaction::getBankDatabase() const
31 {
32 return bankDatabase;
33 } // fim da função getBankDatabase

```

**Figura G.16** Definições de função-membro da classe Transaction.

```

1 // BalanceInquiry.h
2 // Definição da classe BalanceInquiry. Representa uma consulta de saldo.
3 #ifndef BALANCE_INQUIRY_H
4 #define BALANCE_INQUIRY_H
5
6 #include "Transaction.h" // definição da classe Transaction
7
8 class BalanceInquiry : public Transaction
9 {
10 public:
11 BalanceInquiry(int, Screen &, BankDatabase &); // construtor
12 virtual void execute(); // realiza a transação
13 }; // fim da classe BalanceInquiry
14
15 #endif // BALANCE_INQUIRY_H

```

**Figura G.17** Definição da classe BalanceInquiry.

```

1 // BalanceInquiry.cpp
2 // Definições de função-membro da classe BalanceInquiry.
3 #include "BalanceInquiry.h" // definição da classe BalanceInquiry
4 #include "Screen.h" // definição da classe Screen
5 #include "BankDatabase.h" // definição da classe BankDatabase
6
7 // o construtor BalanceInquiry inicializa os membros de dados de classe básica
8 BalanceInquiry::BalanceInquiry(int userAccountNumber, Screen &atmScreen,
9 BankDatabase &atmBankDatabase)
10 : Transaction(userAccountNumber, atmScreen, atmBankDatabase)
11 {
12 // corpo vazio
13 } // fim do construtor de BalanceInquiry
14
15 // realiza transação; sobrescreve a função virtual pura da Transaction
16 void BalanceInquiry::execute()
17 {
18 // obtém as referências ao banco de dados e tela do banco
19 BankDatabase &bankDatabase = getBankDatabase();
20 Screen &screen = getScreen();
21
22 // obtém o saldo disponível da Account do usuário atual
23 double availableBalance =
24 bankDatabase.getAvailableBalance(getAccountNumber());
25
26 // obtém o saldo total da Account do usuário atual
27 double totalBalance =
28 bankDatabase.getTotalBalance(getAccountNumber());
29
30 // exibe as informações sobre o saldo na tela
31 screen.displayMessageLine("\nBalance Information:");
32 screen.displayMessage(" - Available balance: ");
33 screen.displayDollarAmount(availableBalance);
34 screen.displayMessage("\n - Total balance: ");
35 screen.displayDollarAmount(totalBalance);
36 screen.displayMessageLine("");
37 } // fim da função execute

```

**Figura G.18** Definições de função-membro da classe BalanceInquiry.

o saldo na tela do ATM. Lembre-se de que `displayDollarAmount` recebe um argumento `double` e gera a saída dele na tela formatada como uma quantia em dólares. Por exemplo, se `availableBalance` de um usuário for 700.5, a linha 33 gerará a saída de \$700.50. Observe que a linha 36 insere uma linha em branco da saída para separar as informações do saldo da saída subsequente (isto é, o menu principal repetido pela classe ATM depois de executar a classe BalanceInquiry).

## G.11 Classe Withdrawal

A classe `Withdrawal` (figuras G.19–G.20) deriva de `Transaction` e representa uma transação ATM de retirada. A Figura G.19 se expande sobre o arquivo de cabeçalho para aquela classe desenvolvida na Figura 13.31. A classe `Withdrawal` tem um construtor e uma função-membro `execute`, que discutiremos em breve. Lembre-se, no diagrama de classes da Figura 13.29, de que a classe `Withdrawal` tem um atributo, `amount`, o qual a linha 16 implementa como um membro de dados `int`. A Figura 13.28 modela associações entre a classe `Withdrawal` e as classes `Keypad` e `CashDispenser`, para as quais as linhas 17–18 implementam referências `keypad` e `cashDispenser`, respectivamente. A linha 19 é o protótipo de função de uma função utilitária `private` que discutiremos em breve.

### Definições de função-membro da classe *Withdrawal*

A Figura G.20 contém as definições de função-membro da classe `Withdrawal`. A linha 3 inclui (`#includes`) a definição da classe e as linhas 4–7 incluem (`#include`) as definições das outras classes utilizadas nas funções-membro da `Withdrawal`. A linha 11 declara uma

constante global correspondente para a opção de cancelamento no menu de retirada. Discutiremos mais adiante como a classe utiliza essa constante.

O construtor da classe `Withdrawal` (definido nas linhas 13–20 da Figura G.20) tem cinco parâmetros. Ele utiliza um inicializador de classe básica na linha 16 para passar os parâmetros `userAccountNumber`, `atmScreen` e `atmBankDatabase` para o construtor da classe básica `Transaction` configurar os membros de dados que `Withdrawal` herda de `Transaction`. O construtor também aceita as referências `atmKeypad` e `atmCashDispenser` como parâmetros e as atribui para referenciar os membros de dados `keypad` e `cashDispenser` utilizando inicializadores de membro (linha 17).

A classe `Withdrawal` sobreescreve a função virtual `execute` pura da `Transaction` com uma implementação concreta (linhas 23–81) que realiza os passos envolvidos em uma retirada. A linha 25 declara e inicializa uma variável `bool cashDispensed` local. Essa variável indica se cédulas foram entregues (isto é, se a transação foi completada com sucesso) e inicialmente é `false`. A linha 26 declara e inicializa como `false` uma variável `bool transactionCanceled` que indica se a transação foi cancelada pelo usuário. As linhas 29–30 obtêm referências ao banco de dados da instituição financeira e à tela do ATM invocando as funções-membro herdadas da classe básica `Transaction`.

As linhas 33–80 contêm uma instrução `do...while` que executa seu corpo até que o dinheiro seja entregue (isto é, até que `cashDispensed` se torne `true`) ou até que o usuário escolha cancelar (isto é, até que `transactionCanceled` se torne `true`). Utilizamos esse loop continuamente para retornar o usuário ao início da transação se ocorrer um erro (isto é, a quantia de retirada solicitada é maior do que o saldo disponível do usuário ou maior do que a quantia de cédulas no dispensador de cédulas). A linha 36 exibe um menu de quantias de retirada e obtém uma seleção de usuário chamando a função utilitária `private displayMenuOfAmounts` (definida nas linhas 85–129). Essa função-membro exibe o menu das quantias e retorna uma quantia de retirada `int` ou a constante `int CANCELED` para indicar que o usuário optou por cancelar a transação.

A função-membro `displayMenuOfAmounts` (linhas 85–129) primeiro declara a variável local `userChoice` (inicialmente 0) para armazenar o valor que a função-membro retornará (linha 87). A linha 89 obtém uma referência à tela chamando a função-membro `getScreen` herdada da classe básica `Transaction`. A linha 92 declara um array de inteiros das quantias de retirada que correspondem às quantias exibidas no menu de retirada. Ignoramos o primeiro elemento no array (índice 0) porque o menu não tem nenhuma opção 0. A instrução `while` nas linhas 95–126 é repetida até `userChoice` assumir um valor diferente de 0. Veremos mais adiante que isso ocorre quando o usuário faz uma seleção válida no menu. As linhas 98–105 exibem o menu de retirada na tela e solicitam que o usuário insira uma escolha. A linha 107 obtém o input de inteiro pelo teclado. A instrução `switch` nas linhas 110–125 determina como prosseguir com base na entrada do usuário. Se o usuário selecionar um número entre 1 e 5, a linha 117 configura `userChoice` como o valor do elemento em `amounts` no índice `input`. Por exemplo, se o usuário inserir 3 para sacar \$60, a linha 117 configura `userChoice` como o valor de `amounts[ 3 ]` (isto é, 60). A linha 118 termina a instrução `switch`. A variável `userChoice` não é mais igual a 0, assim o `while` nas linhas 95–126 termina e a linha 128 retorna `userChoice`. Se o usuário selecionar a opção cancelar no menu, as linhas 120–121

```

1 // Withdrawal.h
2 // Definição da classe Withdrawal. Representa uma transação de retirada.
3 #ifndef WITHDRAWAL_H
4 #define WITHDRAWAL_H
5
6 #include "Transaction.h" // definição da classe Transaction
7 class Keypad; // declaração antecipada da classe Keypad
8 class CashDispenser; // declaração antecipada da classe CashDispenser
9
10 class Withdrawal : public Transaction
11 {
12 public:
13 Withdrawal(int, Screen &, BankDatabase &, Keypad &, CashDispenser &);
14 virtual void execute(); // realiza a transação
15 private:
16 int amount; // quantia a sacar
17 Keypad &keypad; // referência ao teclado do ATM
18 CashDispenser &cashDispenser; // referência ao dispensador de notas do ATM
19 int displayMenuOfAmounts() const; // exibe o menu de retirada
20 }; // fim da classe Withdrawal
21
22 #endif // WITHDRAWAL_H

```

**Figura G.19** Definição da classe `Withdrawal`.

executam, configurando `userChoice` como `CANCELED` e fazendo com que a função-membro retorne esse valor. Se o usuário não inserir uma seleção válida de menu, as linhas 123–124 exibem uma mensagem de erro e o usuário é retornado ao menu de retirada.

A instrução `if` na linha 39 da função-membro `execute` determina se o usuário selecionou uma quantia de retirada ou escolheu cancelar. Se o usuário cancelar, as linhas 77–78 são executadas para exibir uma mensagem apropriada ao usuário e para configurar `transactionCanceled` como `true`. Isso faz com que o teste de continuação do loop na linha 80 falhe e o controle retorne à função-membro chamadora (isto é, a função-membro `ATM::performTransactions`). Se o usuário escolheu um valor de retirada, a linha 41 atribui a variável local `selection` ao membro de dados `amount`. As linhas 44–45 recuperam o saldo disponível da `Account` do usuário atual e o armazenam em uma variável `double` `availableBalance` local. Em seguida, a instrução `if` na linha 48 determina se a quantia selecionada é menor ou igual ao saldo disponível do usuário. Se não for, as linhas 70–72 exibem uma mensagem de erro apropriada. O controle então continua para o fim do loop `do...while` e este se repete porque tanto `cashDispensed` como `transactionCanceled` ainda são `false`. Se o saldo do usuário for suficientemente alto, a instrução `if` na linha 51 determina se o dispensador de cédulas tem cédulas suficientes para satisfazer a solicitação de retirada invocando a função-membro `isSufficientCashAvailable` de `cashDispenser`. Se essa função-membro retornar `false`, as linhas 64–66 exibem uma mensagem de erro apropriada e o `do...while` se repete. Se houver cédulas suficientes disponíveis, então os requisitos para a retirada serão atendidos e a linha 54 debita `amount` da conta do usuário no banco de dados. As linhas 56–57 então instruem o dispensador de cédulas a entregar as cédulas ao usuário e configuram `cashDispensed` como `true`. Por fim, as linhas 60–61 exibem uma mensagem para o usuário de que as cédulas foram entregues. Como `cashDispensed` agora é `true`, o controle continua depois de `do...while`. Nenhuma instrução adicional aparece abaixo do loop, portanto a função-membro retorna o controle à classe `ATM`.

Note que, nas chamadas de função das linhas 64–66 e 70–72, dividimos o argumento para a função-membro `Screen::displayMessageLine` em dois literais `string`, cada um deles colocado em uma linha separada no programa. Fazemos isso porque cada argumento é muito longo para caber em uma única linha. O C++ concatena (isto é, combina) os literais `string` adjacentes entre si, mesmo se eles estiverem em linhas separadas. Por exemplo, se você escrever `"Happy " "Birthday"` em um programa, o C++ visualizará esses dois literais `string` adjacentes como o único literal `string` `"Happy Birthday"`. Como resultado, quando as linhas 64–66 executarem, `displayMessageLine` receberá uma `string` única como um parâmetro, mesmo que o argumento na chamada de função apareça como dois literais `string`.

## G.12 Classe Deposit

A classe `Deposit` (figuras G.21–G.22) deriva de `Transaction` e representa uma transação ATM de depósito. A Figura G.21 contém a definição de classe `Deposit`. Assim como as classes derivadas `BalanceInquiry` e `Withdrawal`, `Deposit` declara um construtor (linha 13) e a função-membro `execute` (linha 14) — os quais discutiremos daqui a pouco. Lembre-se, a partir do diagrama de classes da Figura 13.29, de que a classe `Deposit` tem um atributo, `amount`, que a linha 16 implementa como um campo `int`. As linhas 17–18 criam membros de dados de referência `keypad` e `depositSlot` que implementam as associações entre a classe `Deposit` e as classes `Keypad` e `DepositSlot` modeladas na Figura 13.28. A linha 19 contém o protótipo de função para uma função utilitária `private::promptForDepositAmount` que discutiremos em breve.

```

1 // Withdrawal.cpp
2 // Definições de função-membro da classe Withdrawal.
3 #include "Withdrawal.h" // definição da classe Withdrawal
4 #include "Screen.h" // definição da classe Screen
5 #include "BankDatabase.h" // definição da classe BankDatabase
6 #include "Keypad.h" // definição da classe Keypad
7 #include "CashDispenser.h" // definição da classe CashDispenser
8
9 // constante global que corresponde à opção de menu para cancelar
10 const static int CANCELED = 6;
11
12 // construtor Withdrawal inicializa os membros de dados da classe
13 Withdrawal::Withdrawal(int userAccountNumber, Screen &atmScreen,
14 BankDatabase &atmBankDatabase, Keypad &atmKeypad,
15 CashDispenser &atmCashDispenser)
16 : Transaction(userAccountNumber, atmScreen, atmBankDatabase),
17 keypad(atmKeypad), cashDispenser(atmCashDispenser)
18 {

```

Figura G.20 Definições de função-membro da classe `Withdrawal`.

(continua)



```

19 // corpo vazio
20 } // fim do construtor de Withdrawal
21
22 // realiza transação; sobrescreve a função virtual da Transaction
23 void Withdrawal::execute()
24 {
25 bool cashDispensed = false; // cédulas ainda não foram entregues
26 bool transactionCanceled = false; // transação ainda não foi cancelada
27
28 // obtém as referências ao banco de dados e à tela do banco
29 BankDatabase &bankDatabase = getBankDatabase();
30 Screen &screen = getScreen();
31
32 // faz um loop até as cédulas serem entregues ou o usuário cancelar
33 do
34 {
35 // obtém a quantia de retirada escolhida pelo usuário
36 int selection = displayMenuOfAmounts();
37
38 // verifica se o usuário escolheu uma quantia de retirada ou cancelou
39 if (selection != CANCELED)
40 {
41 amount = selection; // configura quantia como o valor em dólar selecionado
42
43 // obtém o saldo disponível na conta envolvida
44 double availableBalance =
45 bankDatabase.getAvailableBalance(getAccountNumber());
46
47 // verifica se o usuário tem dinheiro suficiente na conta
48 if (amount <= availableBalance)
49 {
50 // verifica se o dispensador de cédulas tem cédulas suficientes
51 if (cashDispenser.isSufficientCashAvailable(amount))
52 {
53 // atualiza a conta envolvida para refletir a retirada
54 bankDatabase.debit(getAccountNumber(), amount);
55
56 cashDispenser.dispenseCash(amount); // entrega cédulas
57 cashDispensed = true; // as cédulas foram entregues
58
59 // instrui o usuário a pegar as cédulas
60 screen.displayMessageLine(
61 "\nPlease take your cash from the cash dispenser.");
62 } // fim do if
63 else // o dispensador de cédulas não tem cédulas suficientes
64 {
65 screen.displayMessageLine(
66 "\nInsufficient cash available in the ATM."
67 "\n\nPlease choose a smaller amount.");
68 } // fim do if
69 } // fim do if
70 else // não há dinheiro suficiente disponível na conta do usuário
71 {
72 screen.displayMessageLine(
73 "\nInsufficient funds in your account."
74 "\n\nPlease choose a smaller amount.");
75 } // fim do else
76 } // fim do if
77 } // fim do do
78 }

```

**Figura G.20** Definições de função-membro da classe `Withdrawal`.

(continua)



```

75 else // usuário escolheu a opção de menu Cancel
76 {
77 screen.displayMessageLine("\nCanceling transaction...");
78 transactionCanceled = true; // usuário cancelou a transação
79 } // fim do else
80 } while (!cashDispensed && !transactionCanceled); // fim da instrução do...while
81 } // fim da função execute
82
83 // exibe um menu de quantias de saques e a opção para cancelar;
84 // retorna a quantia escolhida ou 0 se o usuário escolher cancelar
85 int Withdrawal::displayMenuOfAmounts() const
86 {
87 int userChoice = 0; // variável local para armazenar o valor de retorno
88
89 Screen &screen = getScreen(); // obtém referência de tela
90
91 // array de quantias que correspondem aos números no menu
92 int amounts[] = { 0, 20, 40, 60, 100, 200 };
93
94 // faz um loop enquanto nenhuma escolha válida for feita
95 while (userChoice == 0)
96 {
97 // exibe o menu
98 screen.displayMessageLine("\nWithdrawal options:");
99 screen.displayMessageLine("1 - $20");
100 screen.displayMessageLine("2 - $40");
101 screen.displayMessageLine("3 - $60");
102 screen.displayMessageLine("4 - $100");
103 screen.displayMessageLine("5 - $200");
104 screen.displayMessageLine("6 - Cancel transaction");
105 screen.displayMessage("\nChoose a withdrawal option (1-6): ");
106
107 int input = keypad.getInput(); // obtém a entrada do usuário pelo teclado
108
109 // determina como prosseguir com base no valor de entrada
110 switch (input)
111 {
112 case 1: // se o usuário escolheu uma quantia de retirada
113 case 2: // (isto é, escolheu a opção 1, 2, 3, 4 ou 5), retorna a
114 case 3: // quantia correspondente do array de quantias
115 case 4:
116 case 5:
117 userChoice = amounts[input]; // salva a escolha do usuário
118 break;
119 case CANCELED: // o usuário escolheu cancelar
120 userChoice = CANCELED; // salva a escolha do usuário
121 break;
122 default: // o usuário não inseriu um valor entre 1 e 6
123 screen.displayMessageLine(
124 "\nInvalid selection. Try again.");
125 } // fim do switch
126 } // fim do while
127
128 return userChoice; // retorna a quantia de retirada ou CANCELADA
129 } // fim da função displayMenuOfAmounts

```

**Figura G.20** Definições de função-membro da classe `Withdrawal`.

(continuação)

```

1 // Deposit.h
2 // Definição da classe Deposit. Representa uma transação de depósito.
3 #ifndef DEPOSIT_H
4 #define DEPOSIT_H
5
6 #include "Transaction.h" // definição da classe Transaction
7 class Keypad; // declaração antecipada da classe Keypad
8 class DepositSlot; // declaração antecipada da classe DepositSlot
9
10 class Deposit : public Transaction
11 {
12 public:
13 Deposit(int, Screen &, BankDatabase &, Keypad &, DepositSlot &);
14 virtual void execute(); // realiza a transação
15 private:
16 double amount; // quantia a depositar
17 Keypad &keypad; // referência ao teclado do ATM
18 DepositSlot &depositSlot; // referência à abertura de depósito do ATM
19 double promptForDepositAmount() const; // obtém a quantia de depósito do usuário
20 }; // fim da classe Deposit
21
22 #endif // DEPOSIT_H

```

**Figura G.21** Definição da classe Deposit.

### Definições de função-membro da classe *Deposit*

A Figura G.22 apresenta o arquivo de implementação da classe *Deposit*. A linha 3 inclui (`#include`) a definição da classe *Deposit* e as linhas 4–7 incluem (`#include`) as definições da classe de outras classes utilizadas em funções-membro da classe *Deposit*. A linha 9 declara uma constante `CANCELED` que corresponde ao valor que um usuário insere para cancelar um depósito. Discutiremos mais adiante como a classe utiliza essa constante.

Como a classe *Withdrawal*, a classe *Deposit* contém um construtor (linhas 12–19) que passa três parâmetros ao construtor da classe básica *Transaction* utilizando um inicializador de classe básica (linha 15). O construtor também tem parâmetros `atmKeypad` e `atmDepositSlot`, que ele atribui a seus membros de dados correspondentes (linha 16).

A função-membro `execute` (linhas 22–62) sobrescreve a função virtual `execute` pura na classe básica *Transaction* com uma implementação concreta que realiza os passos requeridos em uma transação de depósito. As linhas 24–25 obtêm referências para o banco de dados e a tela. A linha 27 solicita para o usuário inserir uma quantia de depósito invocando a função utilitária private `promptForDepositAmount` (definida nas linhas 65–81) e configura o membro de dados `amount` como o valor retornado. A função-membro `promptForDepositAmount` solicita que o usuário insira uma quantia de depósito como um número inteiro de centavos (visto que o teclado numérico do ATM não contém um ponto de fração decimal; isso é compatível com muitos ATMs reais) e retorna o valor `double` que representa a quantia em dólares a ser depositada.

A linha 67 na função-membro `promptForDepositAmount` obtém uma referência à tela do ATM. As linhas 70–71 exibem uma mensagem na tela que solicita ao usuário inserir uma quantia de depósito como um número de centavos ou ‘0’ para cancelar a transação. A linha 72 recebe a entrada do usuário pelo teclado. A instrução `if` nas linhas 75–80 determina se o usuário inseriu uma quantia de depósito real ou optou por cancelar. Se o usuário escolher cancelar, a linha 76 retornará a constante `CANCELED`. Caso contrário, a linha 79 retorna a quantia de depósito depois de converter do número de centavos para uma quantia em dólares fazendo uma coerção de `input` para um `double` e então dividindo por 100. Por exemplo, se o usuário inserir 125 como o número de centavos, a linha 79 retornará 125.0 dividido por 100 ou 1.25 — 125 centavos é \$1.25.

A instrução `if` nas linhas 30–61 da função-membro `execute` determina se o usuário escolheu cancelar a transação em vez de inserir uma quantia de depósito. Se o usuário cancelar, a linha 60 exibe uma mensagem apropriada e a função-membro retorna. Se o usuário inserir uma quantia de depósito, as linhas 33–36 instruem o usuário a inserir um envelope de depósito com a quantia correta. Lembre-se de que a função-membro `Screen displayDollarAmount` gera saída de um `double` formatado como um valor em dólar.

A linha 39 configura uma variável `bool` local com o valor retornado pela função-membro `isEnvelopeReceived` de `depositSlot`, indicando se um envelope de depósito foi recebido. Lembre-se de que codificamos a função-membro `isEnvelopeReceived` (linhas 7–10 da Figura G.10) para retornar sempre `true`, porque estamos simulando a funcionalidade da abertura de depósito e pressupomos que o usuário sempre insere um envelope. Entretanto, codificamos a função-membro `execute` da classe *Deposit* para testar a possibilidade de o usuário não inserir um envelope — a boa engenharia de software exige que os programas sejam responsáveis por todos os possíveis valores de retorno. Portanto, a classe *Deposit* está preparada para futuras versões do `isEnvelopeReceived` que poderiam retornar `false`.

```

1 // Deposit.cpp
2 // Definições de função-membro para a classe Deposit.
3 #include "Deposit.h" // definição da classe Deposit
4 #include "Screen.h" // definição da classe Screen
5 #include "BankDatabase.h" // definição da classe BankDatabase
6 #include "Keypad.h" // definição da classe Keypad
7 #include "DepositSlot.h" // definição da classe DepositSlot
8
9 const static int CANCELED = 0; // constante representando a opção de cancelamento
10
11 // o construtor Deposit inicializa os membros de dados da classe
12 Deposit::Deposit(int userAccountNumber, Screen &atmScreen,
13 BankDatabase &atmBankDatabase, Keypad &atmKeypad,
14 DepositSlot &atmDepositSlot)
15 : Transaction(userAccountNumber, atmScreen, atmBankDatabase),
16 keypad(atmKeypad), depositSlot(atmDepositSlot)
17 {
18 // corpo vazio
19 } // fim do construtor de Deposit
20
21 // realiza transação; sobrescreve a função virtual pura da Transaction
22 void Deposit::execute()
23 {
24 BankDatabase &bankDatabase = getBankDatabase(); // obtém a referência
25 Screen &screen = getScreen(); // obtém a referência
26
27 amount = promptForDepositAmount(); // obtém a quantia de depósito do usuário
28
29 // verifica se o usuário inseriu uma quantia de depósito ou cancelou
30 if (amount != CANCELED)
31 {
32 // solicita o envelope de depósito contendo a quantia especificada
33 screen.displayMessage(
34 "\nPlease insert a deposit envelope containing ");
35 screen.displayDollarAmount(amount);
36 screen.displayMessageLine(" in the deposit slot.");
37
38 // recebe o envelope de depósito
39 bool envelopeReceived = depositSlot.isEnvelopeReceived();
40
41 // verifica se o envelope de depósito foi recebido
42 if (envelopeReceived)
43 {
44 screen.displayMessageLine("\nYour envelope has been received."
45 "\nNOTE: The money just will not be available until we"
46 "\nverify the amount of any enclosed cash, and any enclosed "
47 "checks clear.");
48
49 // credita na conta para refletir o depósito
50 bankDatabase.credit(getAccountNumber(), amount);
51 } // fim do if
52 else // o envelope de depósito não foi recebido
53 {
54 screen.displayMessageLine("\nYou did not insert an "
55 "envelope, so the ATM has canceled your transaction.");
56 } // fim do else

```

Figura G.22 Definições de função-membro da classe Deposit.

(continua)

```

57 } // fim do if
58 else // o usuário cancelou em vez de inserir uma quantia
59 {
60 screen.displayMessageLine("\nCanceling transaction...");
61 } // fim do else
62 } // fim da função execute
63
64 // solicita que o usuário insira uma quantia de depósito em centavos
65 double Deposit::promptForDepositAmount() const
66 {
67 Screen &screen = getScreen(); // obtém a referência à tela
68
69 // exibe o prompt e recebe a entrada
70 screen.displayMessage("\nPlease enter a deposit amount in "
71 "CENTS (or 0 to cancel): ");
72 int input = keypad.getInput(); // recebe a entrada da quantia do depósito
73
74 // verifica se o usuário cancelou ou inseriu uma quantia válida
75 if (input == CANCELED)
76 return CANCELED;
77 else
78 {
79 return static_cast< double >(input) / 100; // retorna a quantia em dólares
80 } // fim do else
81 } // fim da função promptForDepositAmount

```

**Figura G.22** Definições de função-membro da classe `Deposit`.

(continuação)

As linhas 44–50 executam se a abertura para depósito receber um envelope. As linhas 44–47 exibem uma mensagem apropriada para o usuário. A linha 50 então credita a quantia de depósito na conta do usuário no banco de dados. As linhas 54–55 executarão se a abertura de depósito não receber um envelope de depósito. Nesse caso, exibimos uma mensagem ao usuário indicando que o ATM cancelou a transação. A função-membro então retorna sem modificar a conta do usuário.

## G.13 Programa de teste `ATMCaseStudy.cpp`

`ATMCaseStudy.cpp` (Figura G.23) é um programa C++ simples que permite iniciar, ou ‘ligar’, o ATM e testar a implementação de nosso modelo de sistema ATM. A função `main` do programa (linhas 6–11) nada mais faz que instanciar um novo objeto ATM chamado `atm` (linha 8) e invocar sua função-membro `run` (linha 9) para iniciar o ATM.

```

1 // ATMCaseStudy.cpp
2 // Programa driver para o estudo de caso do ATM.
3 #include "ATM.h" // definição da classe ATM
4
5 // a função main cria e executa o ATM
6 int main()
7 {
8 ATM atm; // cria um objeto ATM
9 atm.run(); // instrui o ATM a iniciar
10 return 0;
11 } // fim do main

```

**Figura G.23** `ATMCaseStudy.cpp` inicia o sistema ATM.

## **G.14** Síntese

Parabéns por ter completado todo o estudo de caso de engenharia de software do ATM! Esperamos que essa experiência tenha sido valiosa e tenha reforçado muitos conceitos que você aprendeu nos capítulos 1 a 10. Gostaríamos, sinceramente, de receber seus comentários, críticas e sugestões. Você pode nos contatar pelo endereço de correio eletrônico [dei1@dei1.com](mailto:dei1@dei1.com). Responderemos prontamente.



# UML 2: tipos de diagramas adicionais

## H.1 Introdução

Se você já leu as seções opcionais de “Estudo de caso de engenharia de software” nos capítulos 2–7, 9 e 13, deve ter agora um entendimento satisfatório dos tipos de diagrama UML que utilizamos para modelar nosso sistema ATM. O estudo de caso é concebido para uso no primeiro ou segundo semestre dos cursos, portanto limitamos a nossa discussão a um subconjunto conciso da UML. A UML 2 fornece um total de 13 tipos de diagramas. O final da Seção 2.8 resume os seis tipos de diagramas que utilizamos no estudo de caso. Este apêndice lista e define brevemente os sete tipos de diagramas restantes.

## H.2 Tipos de diagramas adicionais

A seguir estão os sete tipos de diagramas que optamos por não utilizar no nosso “Estudo de caso de engenharia de software”.

- Os **diagramas de objetos** modelam um ‘instantâneo’ do sistema modelando os objetos de um sistema e seus relacionamentos em um momento específico. Cada objeto representa uma instância de uma classe proveniente de um diagrama de classes; e diversos objetos podem ser criados a partir de uma única classe. Para nosso sistema ATM, um diagrama de objetos poderia mostrar lado a lado vários objetos Account distintos, ilustrando o fato de que todos eles fazem parte das contas no banco de dados da instituição financeira.
- Os **diagramas de componentes** modelam **artefatos** e **componentes** — os recursos (que incluem arquivos-fonte) — que compõem o sistema.
- Os **diagramas de implantação** modelam os requisitos de tempo de execução do sistema (tais como o computador ou computadores em que o sistema residirá), requisitos de memória ou outros dispositivos que o sistema requer durante a execução.
- Os **diagramas de pacotes** modelam a estrutura hierárquica dos **pacotes** (que são grupos de classes) no sistema em tempo de compilação e os relacionamentos que existem entre os pacotes.
- Os **diagramas de estrutura compostos** modelam a estrutura interna de um objeto complexo em tempo de execução. Novos na UML 2, esses diagramas permitem aos designers de sistema decompor hierarquicamente um objeto complexo em partes menores. Os diagramas de estrutura compostos estão além do escopo do nosso estudo de caso. Eles são mais apropriados a aplicativos industriais maiores, que exibem complexos agrupamentos de objetos em tempo de execução.
- Os **diagramas de visão geral das interações**, novos na UML 2, fornecem um resumo do fluxo de controle no sistema combinando elementos de vários tipos dos diagramas comportamentais (por exemplo, diagramas de atividades, diagramas de seqüências).

- Os **diagramas de sincronização**, também novos na UML 2, modelam as restrições de sincronização impostas na etapa das alterações e interações entre objetos em um sistema.

Para aprender mais sobre esses diagramas e tópicos UML avançados, visite [www.uml.org](http://www.uml.org) e os recursos da Web listados no final das seções 1.17 e 2.8.



# Recursos sobre C++ na Internet e na Web

Este apêndice contém uma lista de recursos C++ que estão disponíveis na Internet e na World Wide Web. Esses recursos incluem FAQs (perguntas feitas com frequência), tutoriais, links para o padrão ANSI/ISO C++, informações sobre compiladores C++ populares e acesso a recursos livres/gratuitos como compiladores, demos, livros, tutoriais, ferramentas de software, artigos, entrevistas, conferências, periódicos e revistas, cursos on-line, newsgroups e recursos de carreira. Para obter informações adicionais sobre o American National Standards Institute (ANSI) e suas atividades relacionadas com C++, visite [www.ansi.org](http://www.ansi.org).

## I.1 Recursos

[www.cplusplus.com](http://www.cplusplus.com)

Este site contém informações sobre a história e o desenvolvimento do C++, bem como tutoriais, documentação, material de referência, código-fonte e fóruns.

[www.possibility.com/Cpp/CppCodingStandard.html](http://www.possibility.com/Cpp/CppCodingStandard.html)

O site C++ *Coding Standard* examina o padrão e o processo de padronização do C++. O site inclui tópicos como imposição de padrões, formatação, portabilidade e documentação, e oferece links para recursos C++ adicionais da Web.

[http://www.research.att.com/~bs/bs\\_faq2.html](http://www.research.att.com/~bs/bs_faq2.html)

A FAQ sobre o estilo e a técnica C++ de Bjarne Stroustrup, o criador da linguagem, fornece respostas a perguntas comuns sobre C++.

[help-site.com/cpp.html](http://help-site.com/cpp.html)

O *Help-site.com* fornece links para recursos C++ na Web, incluindo tutoriais e uma FAQ sobre o C++.

[www.glenmcc1.com/tutor.htm](http://www.glenmcc1.com/tutor.htm)

Este site de referência discute tópicos como projeto orientado a objetos e escrita de código robusto. O site fornece introduções aos tópicos da linguagem C++, incluindo a palavra-chave `static`, o tipo de dados `bool`, namespaces, a Standard Template Library e a alocação de memória.

[www.programmersheaven.com/zone3](http://www.programmersheaven.com/zone3)

Este site fornece links para artigos, tutoriais, ferramentas de desenvolvimento, uma extensa coleção de bibliotecas e código-fonte C++ gratuita.

[www.ha19k.com/cug](http://www.ha19k.com/cug)

O site *C/C++ Users Group (CUG)* contém recursos, periódicos, shareware e freeware do C++.

[www.devx.com](http://www.devx.com)

O *DevX* é um recurso abrangente para programadores que fornece as últimas notícias, ferramentas e técnicas para várias linguagens de programação. O *C++ Zone* oferece dicas, fóruns de discussão, ajuda técnica e boletins on-line.

[www.cprogramming.com](http://www.cprogramming.com)

Este site contém tutoriais interativos, questionários, artigos, periódicos, downloads de compiladores, recomendações de livros e código-fonte gratuito.

[www.acm.org/crossroads/xrds3-2/ovp32.html](http://www.acm.org/crossroads/xrds3-2/ovp32.html)

O site *Association for Computing Machinery (ACM)* oferece uma abrangente listagem de recursos relacionados com C++, incluindo textos recomendados, periódicos e revistas, publicação de padrões, boletins, FAQs e newsgroups.

[www.comeaucomputing.com/resources](http://www.comeaucomputing.com/resources)

O site *Comeau Computing's* tem links para discussões técnicas, FAQs (incluindo uma FAQ dedicada aos templates), grupos de usuários, newsgroups e um compilador C++ on-line.

[www.exciton.cs.rice.edu/CppResources](http://www.exciton.cs.rice.edu/CppResources)

O site fornece um documento que resume os aspectos técnicos do C++. O site também discute as diferenças entre Java e C++.

[www.accu.informika.ru/resources/public/terse/cpp.htm](http://www.accu.informika.ru/resources/public/terse/cpp.htm)

O site *The Association of C & C++ Users (ACCU)* contém links para tutoriais, artigos, informações de desenvolvedor, discussões e resenhas de livros sobre C++.

[www.cuj.com](http://www.cuj.com)

O *C/C++ User's Journal* é uma revista on-line que contém artigos, tutoriais e downloads. O site apresenta notícias sobre C++, fóruns e links para informações sobre ferramentas de desenvolvimento.

[directory.google.com/Top/Computers/Programming/Languages/C++/Resources/Directories](http://directory.google.com/Top/Computers/Programming/Languages/C++/Resources/Directories)

O diretório de recursos sobre o C++ do Google classifica os sites mais úteis sobre o C++.

[www.compinfo-center.com/c++.htm](http://www.compinfo-center.com/c++.htm)

Este site fornece links para FAQs, newsgroups e revistas sobre o C++.

[www.apl.jhu.edu/~paulmac/c++-references.html](http://www.apl.jhu.edu/~paulmac/c++-references.html)

Este site contém resenhas de livros e recomendações para programadores em C++ de nível básico, intermediário e avançado e links para recursos C++ on-line, incluindo livros, revistas e tutoriais.

[www.cmcrossroads.com/bradapp/links/cplusplus-links.html](http://www.cmcrossroads.com/bradapp/links/cplusplus-links.html)

Esse site divide os links em categorias, incluindo Recursos e Diretórios, Grupos de Projetos e Grupos de Trabalho, bibliotecas, Treinamento, Tutoriais, Publicações e Convenções de Codificação.

[www.codeproject.com](http://www.codeproject.com)

Os artigos, trechos de código, discussões de usuário, livros e notícias sobre programação em C++, C# e .NET estão disponíveis neste site.

[www.austinlinks.com/CPlusPlus](http://www.austinlinks.com/CPlusPlus)

O site *Quadralay Corporation's* tem links para numerosos recursos sobre o C++, incluindo as bibliotecas Visual C++/MFC, C++, informações sobre programação, recursos de carreira em C++ e uma lista de tutoriais e outras ferramentas on-line para aprender C++.

[www.csci.csusb.edu/dick/c++std](http://www.csci.csusb.edu/dick/c++std)

Os links para o ANSI/ISO C++ Standard e o grupo Usenet comp.std.c++ estão disponíveis neste site.

[www.research.att.com/~bs/homepage.html](http://www.research.att.com/~bs/homepage.html)

Esta é a home page de Bjarne Stroustrup, o criador da linguagem de programação C++. Este site fornece uma lista de recursos do C++, FAQs e outras informações C++ úteis.

## I.2 Tutoriais

[www.cprogramming.com/tutorial.html](http://www.cprogramming.com/tutorial.html)

Este site oferece um tutorial passo a passo, com código de exemplo, que abrange E/S de arquivo, recursão, árvores binárias, classes de template e mais.

[www.programmersheaven.com/zone3/cat34](http://www.programmersheaven.com/zone3/cat34)

Os tutoriais gratuitos que são apropriados a muitos níveis de habilidade estão disponíveis neste site.

[www.programmershelp.co.uk/c%2B%2Btutorials.php](http://www.programmershelp.co.uk/c%2B%2Btutorials.php)

Este site contém cursos on-line gratuitos e uma lista abrangente de tutoriais sobre C++. Este site também fornece FAQs, downloads e outros recursos.

[www.codeproject.com/script/articles/beginners.asp](http://www.codeproject.com/script/articles/beginners.asp)

Este site lista tutoriais e artigos disponíveis para iniciantes em C++.

[www.eng.hawaii.edu/Tutor/Make](http://www.eng.hawaii.edu/Tutor/Make)

Este site fornece um tutorial que descreve como criar makefiles.

[www.cpp-home.com](http://www.cpp-home.com)

Tutoriais gratuitos, discussões, salas de bate-papo, artigos, compiladores, fóruns e questionários on-line relacionados ao C++ estão disponíveis neste site. Os tutoriais sobre C++ abrangem tópicos como ActiveX/COM, MFC e imagens gráficas.

[www.codebeach.com](http://www.codebeach.com)

O *Code Beach* contém código-fonte, tutoriais, livros e links para linguagens de programação importantes, incluindo C++, Java, ASP, Visual Basic, XML, Python, Perl e C#.

[www.kegel.com/academy/tutorials.html](http://www.kegel.com/academy/tutorials.html)

Este site fornece links para tutoriais em C, C++ e linguagens assembly.

## I.3 FAQs

[www.faqs.org/faqs/by-newsgroup/comp/comp.lang.c++.html](http://www.faqs.org/faqs/by-newsgroup/comp/comp.lang.c++.html)

Este site consiste em links para FAQs e tutoriais coletados do newsgroup Comp.Lang.C++.

[www.eskimo.com/~scs/C-faq/top.html](http://www.eskimo.com/~scs/C-faq/top.html)

Esta lista de FAQ C contém tópicos como ponteiros, alocação de memória e strings.

[www.technion.ac.il/technion/tcc/usg/Ref/C\\_Programming.html](http://www.technion.ac.il/technion/tcc/usg/Ref/C_Programming.html)

Este site contém referências de programação C/C++, incluindo FAQs e tutoriais.

[www.faqs.org/faqs/by-newsgroup/comp/comp.compilers.html](http://www.faqs.org/faqs/by-newsgroup/comp/comp.compilers.html)

Este site contém uma lista de FAQs geradas no newsgroup comp.compilers.

## I.4 Visual C++

[msdn.microsoft.com/visualc](http://msdn.microsoft.com/visualc)

A página Visual C++ da Microsoft fornece informações sobre os últimos lançamentos do Visual C++ .NET.

[www.freeprogrammingresources.com/visualcpp.html](http://www.freeprogrammingresources.com/visualcpp.html)

Este site contém recursos de programação gratuitos para programadores em Visual C++, incluindo tutoriais e aplicativos de programação de exemplo.

[www.mvps.org/vcfaq](http://www.mvps.org/vcfaq)

O site *Most Valuable Professional (MVP)* contém uma FAQ sobre o Visual C++.

[www.onesmartclick.com/programming/visual-cpp.html](http://www.onesmartclick.com/programming/visual-cpp.html)

Este site contém tutoriais, livros on-line, dicas, truques, FAQs e depuração sobre o Visual C++.

## I.5 Newsgroups

[ai.kaist.ac.kr/~ymkim/Program/c++.html](http://ai.kaist.ac.kr/~ymkim/Program/c++.html)

Este site oferece tutoriais, bibliotecas, compiladores populares, FAQs e newsgroups, incluindo comp.lang.c++.

[www.coding-zone.co.uk/cpp/cnewsgroups.shtml](http://www.coding-zone.co.uk/cpp/cnewsgroups.shtml)

Esse site inclui links para vários newsgroups C++ incluindo comp.lang.c, comp.lang.c++ e comp.lang.c++.moderated, para citar alguns.

## I.6 Compiladores e ferramentas de desenvolvimento

[msdn.microsoft.com/visualc](http://msdn.microsoft.com/visualc)

O site *Microsoft Visual C++* fornece informações de produto, resumos, materiais suplementares e informações sobre pedidos do compilador Visual C++.

[lab.msdn.microsoft.com/express/visualc/](http://lab.msdn.microsoft.com/express/visualc/)

Você pode fazer download do Microsoft Visual C++ Express Beta gratuitamente neste site Web.

[msdn.microsoft.com/visualc/vctoolkit2003/](http://msdn.microsoft.com/visualc/vctoolkit2003/)

Visite este site para fazer download do Visual C++ Toolkit 2003.

[www.borland.com/bcppbuilder](http://www.borland.com/bcppbuilder)

Este é um link para o *Borland C++ Builder 6*. Uma versão de linha de comando livre está disponível para download.

[www.thefreecountry.com/developercity/ccompilers.shtml](http://www.thefreecountry.com/developercity/ccompilers.shtml)

Este site lista compiladores C e C++ gratuitos para uma variedade de sistemas operacionais.

[www.faqs.org/faqs/by-newsgroup/comp/comp.compilers.html](http://www.faqs.org/faqs/by-newsgroup/comp/comp.compilers.html)

Este site lista as FAQs geradas dentro do newsgroup comp.compilers.

[www.compilers.net/Dir/Free/Compilers/CCpp.htm](http://www.compilers.net/Dir/Free/Compilers/CCpp.htm)

*Compilers.net* foi projetado para ajudar usuários a localizar compiladores.

[developer.intel.com/software/products/compilers/cwin/index.htm](http://developer.intel.com/software/products/compilers/cwin/index.htm)

O *Intel® C++ Compiler 8.1 for Windows* está disponível neste site.

[www.intel.com/software/products/compilers/clinux/index.htm](http://www.intel.com/software/products/compilers/clinux/index.htm)

O *Intel® C++ Compiler 8.1 for Linux* está disponível neste site.

[www.symbian.com/developer/development/cppdev.html](http://www.symbian.com/developer/development/cppdev.html)

O Symbian fornece um C++ Developer's Pack e links para vários recursos, incluindo código e ferramentas de desenvolvimento para programadores em C++ (particularmente aqueles que trabalham com o sistema operacional Symbian).

[www.gnu.org/software/gcc/gcc.html](http://www.gnu.org/software/gcc/gcc.html)

O site *GNU Compiler Collection (GCC)* inclui links para fazer download de compiladores GNU para C++, C, Objective C e outras linguagens.

[www.bloodshed.net/devcpp.html](http://www.bloodshed.net/devcpp.html)

O *Bloodshed Dev-C++* é um ambiente de desenvolvimento integrado livre para C++.

### *Fornecedores independentes que fornecem bibliotecas para cálculos financeiros precisos*

[www.roguewave.com/products/sourcepro/analysis/](http://www.roguewave.com/products/sourcepro/analysis/)

As bibliotecas SourcePro Analysis do RogueWave incluem classes para cálculos monetários precisos, análise de dados e algoritmos matemáticos essenciais.

[www.boic.com/numorder.htm](http://www.boic.com/numorder.htm)

A classe Bas/1 Number da Base One International Corporation implementa cálculos matemáticos altamente precisos.

## **I.7 Standard Template Library**

### *Tutoriais*

[www.cs.brown.edu/people/jak/programming/stl-tutorial/tutorial.html](http://www.cs.brown.edu/people/jak/programming/stl-tutorial/tutorial.html)

Este tutorial de STL é organizado por exemplos, filosofia, componentes e extensão de STL. Você encontrará exemplos de código que utilizam os componentes STL, explicações úteis e diagramas úteis.

[www.xraylith.wisc.edu/~khan/software/stl/os\\_examples/examples.html](http://www.xraylith.wisc.edu/~khan/software/stl/os_examples/examples.html)

Este site é útil simplesmente para o aprendizado de STL. Você encontrará uma introdução à STL e exemplos do ObjectSpace STL Tool Kit.

[cplus.about.com/od/stltutorial/1/b1stl.htm](http://cplus.about.com/od/stltutorial/1/b1stl.htm)

Este tutorial discute todos os recursos STL.

[www.yolinux.com/TUTORIALS/LinuxTutorialC++STL.html](http://www.yolinux.com/TUTORIALS/LinuxTutorialC++STL.html)

Este site com o foco em Linux inclui um tutorial de STL e exemplos.

[www.cs.rpi.edu/~musser/stl-book](http://www.cs.rpi.edu/~musser/stl-book)

O site RPI STL inclui informações sobre como a STL difere de outras bibliotecas C++ e sobre como compilar programas que utilizam a STL. Uma lista de STL inclui arquivos, programas de exemplo que utilizam STL, classes contêineres STL e categorias de iterador STL que estão disponíveis. O site também fornece uma lista de compiladores compatíveis com STL, sites de FTP para código-fonte STL e materiais relacionados.

### *Referências*

[www.sgi.com/tech/stl](http://www.sgi.com/tech/stl)

O Silicon Graphics Standard Template Library Programmer's Guide é um recurso útil para obter informações sobre a STL. Você pode fazer download do código-fonte STL neste site e encontrar as últimas informações, documentação de design e links para outros recursos STL.

[www.byte.com/art/9510/sec12/art3.htm](http://www.byte.com/art/9510/sec12/art3.htm)

O site *Byte Magazine* tem a cópia de um artigo escrito por um dos criadores da Standard Template Library, Alexander Stepanov, que fornece informações sobre o uso da STL na programação genérica.

### *ANSI/ISO C++ Standard*

[www.ansi.org](http://www.ansi.org)

Você pode comprar uma cópia do C++ Standard neste site.



*Ler as entrelinhas foi mais fácil  
que seguir o texto.*  
Aristófanes

*Sim; das tábuas da memória  
hei de todas as notícias frívolas  
apagar...*  
William Shakespeare

# Introdução à XHTML

## OBJETIVOS

Neste apêndice, você aprenderá:

- O que são componentes importantes dos documentos XHTML.
- Como utilizar XHTML para criar páginas Web.
- A adicionar imagens a páginas Web.
- Como criar e utilizar hyperlinks para navegar nas páginas Web.
- Como marcar listas de informações.
- A criar formulários.

- J.1** Introdução
- J.2** Editando XHTML
- J.3** Primeiro exemplo de XHTML
- J.4** Cabeçalhos
- J.5** Vinculando
- J.6** Imagens
- J.7** Caracteres especiais e mais quebras de linha
- J.8** Listas não ordenadas
- J.9** Listas aninhadas e listas ordenadas
- J.10** Tabelas XHTML básicas
- J.11** Tabelas e formatação XHTML intermediárias
- J.12** Formulários XHTML básicos
- J.13** Formulários XHTML mais complexos
- J.14** Recursos na Internet e na Web

Resumo | Terminologia

## J.1 Introdução

Neste apêndice, introduzimos **XHTML**<sup>1</sup> — a **Extensible Hypertext Markup Language** para criar conteúdo Web. Diferentemente das linguagens de programação procedurais, como C, Fortran, Cobol e Visual Basic, a XHTML é uma **linguagem de marcação** que especifica o formato do texto que é exibido em um navegador Web, como o Microsoft Internet Explorer ou Netscape Communicator.

Uma questão-chave ao utilizar a XHTML é a separação da **apresentação de um documento** (isto é, a aparência do documento quando renderizado por um navegador) da estrutura das informações do documento. Por todo este apêndice, discutiremos essa questão em profundidade.

Neste apêndice, construímos várias páginas Web completas destacando texto, hyperlinks, imagens, linhas horizontais e quebras de linha. Discutimos também recursos XHTML mais substanciais, incluindo a apresentação de informações em tabelas e a incorporação de formulários para coletar informações de um visitante da página Web. No final deste apêndice, você se familiarizará com os recursos XHTML mais comumente utilizados e será capaz de criar documentos Web mais complexos. Neste apêndice, não apresentamos nenhuma programação em C++.

## J.2 Editando XHTML

Neste apêndice, escrevemos XHTML na sua **forma de código-fonte**. Criamos **documentos XHTML** digitando-os com um editor de textos (por exemplo, Bloco de Notas, WordPad, vi ou emacs) e salvando os documentos com uma extensão de nome de arquivo **.html** ou **.htm**.



### Boa prática de programação J.1

*Atribua aos documentos nomes de arquivo que descrevam sua funcionalidade. Essa prática pode ajudá-lo a identificar documentos mais rapidamente. Também ajuda as pessoas que querem estabelecer um link para uma página, atribuindo-lhes um nome fácil de lembrar. Por exemplo, se você estiver escrevendo um documento XHTML que contém as informações de produto, pode querer chamá-la de `products.html`.*

Máquinas que executam softwares especializados, chamadas de **servidor Web**, armazenam documentos XHTML. Os clientes (por exemplo, navegadores Web) solicitam recursos específicos, como documentos XHTML, a partir do servidor Web. Por exemplo, digitar `www.deitel.com/books/downloads.htm` no campo de endereço de um navegador Web solicita `downloads.htm` a partir do servidor Web que executa em `www.deitel.com`. Esse documento está localizado em um diretório de nome `books`.

## J.3 Primeiro exemplo de XHTML

Neste apêndice, apresentamos a marcação XHTML e fornecemos capturas de tela que mostram como o Internet Explorer renderiza (isto é, exibe) a XHTML. Cada documento XHTML que mostramos tem números de linha para a conveniência do leitor. Esses números de linha não fazem parte dos documentos XHTML.

<sup>1</sup> A XHTML substituiu a Hypertext Markup Language (HTML) como o principal meio de descrever o conteúdo Web. A XHTML fornece recursos mais robustos, mais ricos e mais extensíveis do que a HTML. Para informações adicionais sobre XHTML/HTML, visite [www.w3.org/markup](http://www.w3.org/markup).

Nosso primeiro exemplo (Figura J.1) é um documento XHTML chamado `main.html` que exibe a mensagem `Welcome to XHTML!` no navegador. A linha-chave no programa é a linha 14, que instrui o navegador a exibir `Welcome to XHTML!` Agora vamos considerar cada linha do programa.

As linhas 1–3 são requeridas em documentos XHTML para se adaptarem à sintaxe XHTML adequada. As linhas 5–6 são **comentários XHTML**. Os criadores do documento XHTML inserem comentários para melhorar a legibilidade da marcação e para descrever o conteúdo de um documento. Os comentários também ajudam outras pessoas a ler e entender a marcação e o conteúdo de um documento XHTML. Os comentários não fazem com que o navegador realize nenhuma ação quando o usuário carrega o documento XHTML no navegador Web para visualizá-lo. Os comentários XHTML sempre iniciam com `<!--` e terminam com `-->`. Cada um de nossos exemplos de XHTML inclui comentários que especificam o número da figura e o nome de arquivo e fornecem uma breve descrição do propósito do exemplo. Exemplos subsequentes incluem comentários na marcação, especialmente para destacar novos recursos.



## Boa prática de programação J.2

*Coloque comentários em toda a sua marcação. Os comentários ajudam outros programadores a entender a marcação, auxiliam na depuração e listam informações úteis que você não quer que o navegador renderize. Os comentários também o ajudam a entender sua própria marcação quando você abrir futuramente um documento para fazer modificações ou atualizações.*

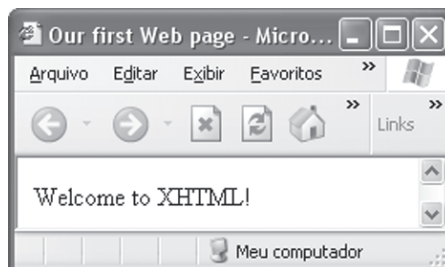
A marcação XHTML contém texto que representa o conteúdo de um documento e os **elementos** que especificam a estrutura de um documento. Alguns elementos importantes de um documento XHTML incluem o **elemento `html`**, o **elemento `head`** e o **elemento `body`**. O elemento `html` inclui a **seção cabeçalho** (representada pelo elemento `head`) e a **seção corpo** (representada pelo elemento `body`). A seção cabeçalho contém informações sobre o documento XHTML, como o **título** do documento. A seção cabeçalho também pode conter instruções de formatação do documento especiais chamadas de **folhas de estilo** e programas do lado do cliente chamados de **scripts** para criar páginas Web dinâmicas. A seção corpo contém o conteúdo da página que o navegador exibe quando o usuário visita a página Web.

Os documentos XHTML delimitam um elemento com tags **iniciais** e  **finais**. Um tag inicial consiste no nome de elemento entre colchetes angulares (por exemplo, `<html>`). Um tag final consiste no nome de elemento precedido por um `/` entre colchetes angulares (por exemplo, `</html>`). Neste exemplo, as linhas 8 e 16 definem o começo e o final do elemento `html`. Observe que o tag final na linha 16 tem o mesmo nome que o tag inicial, mas é precedido por um `/` dentro dos colchetes angulares. Muitos tags iniciais definem **atributos** que fornecem informações adicionais sobre um elemento. Os navegadores podem utilizar essas informações adicionais para determinar

```

1 <?xml version = "1.0"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5 <!-- Figura J.1: main.html -->
6 <!-- Nossa primeira página Web. -->
7
8 <html xmlns = "http://www.w3.org/1999/xhtml">
9 <head>
10 <title>Our first Web page</title>
11 </head>
12
13 <body>
14 <p>Welcome to XHTML!</p>
15 </body>
16 </html>

```



**Figura J.1** Primeiro exemplo de XHTML.



como processar o elemento. Cada atributo tem um **nome** e um **valor**, separado por um sinal igual (=). A linha 8 especifica um atributo (xmlns) e o valor requerido (<http://www.w3.org/1999/xhtml>) pelo elemento html em um documento XHTML.



### Erro comum de programação J.1

*Não incluir valores de atributo entre aspas simples ou duplas é um erro de sintaxe.*



### Erro comum de programação J.2

*Utilizar letras maiúsculas em um elemento XHTML ou nome de atributo é um erro de sintaxe.*

Um documento XHTML divide o elemento html em duas seções — cabeçalho e corpo. As linhas 9–11 definem a seção cabeçalho da página Web com um elemento head. A linha 10 especifica um elemento title. Este é chamado **elemento aninhado**, porque está incluído entre os tags inicial e final do elemento head. O elemento head também é um elemento aninhado, porque está incluído entre os tags inicial e final do elemento html. O elemento **title** descreve a página Web. Os títulos normalmente aparecem na **barra de título** na parte superior da janela do navegador e também como o texto que identifica uma página quando os usuários adicionam a página à lista Favorites ou Bookmarks, que permitem aos usuários retornar a seus sites favoritos. Os sistemas de pesquisa (isto é, sites que permitem aos usuários pesquisar a Web) também utilizam o title para propósitos de catalogação.



### Boa prática de programação J.3

*Recuar elementos aninhados enfatiza a estrutura de um documento e promove a legibilidade.*



### Erro comum de programação J.3

*A XHTML não permite que os tags se sobreponham — um tag final de elemento aninhado deve aparecer no documento antes do tag final do elemento envolvente. Por exemplo, os tags XHTML aninhados `<head><title>hello</head></title>` produzem um erro de sintaxe, porque o tag `</head>` final do elemento head envolvente aparece antes do tag `</title>` final do elemento title aninhado.*



### Boa prática de programação J.4

*Utilize uma convenção para atribuição de nomes title consistente para todas as páginas em um site. Por exemplo, se um site é chamado de “Bailey’s Web Site”, então o title da página principal poderia ser “Bailey’s Web Site — Links”. Essa prática pode ajudar os usuários a entender melhor a estrutura do site Web.*

A linha 13 abre o elemento body do documento. A seção corpo de um documento XHTML especifica o conteúdo do documento, que pode incluir texto e tags.

Alguns tags, como os **tags de parágrafo** (`<p>` e `</p>`) na linha 14, marcam o texto para exibição em um navegador. Todo texto colocado entre os tags `<p>` e `</p>` forma um parágrafo. Quando o navegador renderizar um parágrafo, uma linha em branco normalmente aparece antes e depois do texto de um parágrafo.

Esse documento acaba com dois tags de fechamento (linhas 22–23). Esses tags fecham os elementos body e html, respectivamente. O tag `</html>` final em um documento XHTML informa o navegador de que a marcação XHTML está completa.

Para ver esse exemplo no Internet Explorer, realize os seguintes passos:

1. Copie os exemplos do Apêndice J para a sua máquina (esses exemplos estão disponíveis no CD-ROM que acompanha este livro).
2. Carregue o Internet Explorer e selecione Abrir... a partir do menu Arquivo. Isso exibe a caixa de diálogo Abrir.
3. Clique no botão Procurar... da caixa de diálogo Abrir para exibir a caixa de diálogo de arquivo Microsoft Internet Explorer.
4. Navegue para o diretório que contém os exemplos do Apêndice J e selecione o arquivo main.html; então clique em Abrir.
5. Clique em OK para fazer o Internet Explorer (ou qualquer outro navegador) exibir o documento. Outros exemplos são abertos de maneira semelhante.

Nesse ponto, a janela do navegador deve ser semelhante à captura de tela do exemplo mostrado na Figura J.1.

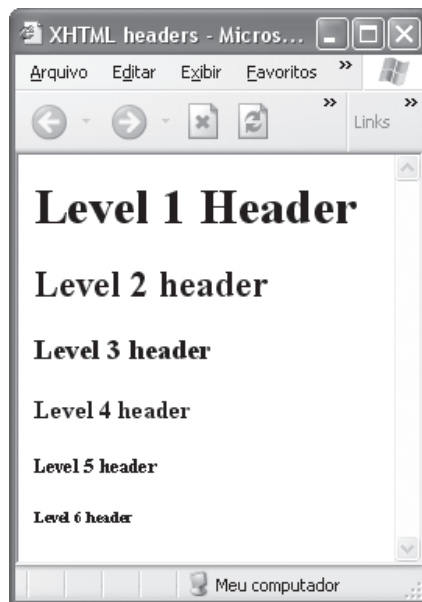
## J.4 Cabeçalhos

Algum texto em um documento XHTML pode ser mais importante que outro. Por exemplo, o texto nesta seção é considerado mais importante do que uma nota de rodapé. A XHTML fornece seis **níveis de título** (**headers**), chamados **elementos de nível de título**, para especificar a importância relativa das informações. A Figura J.2 demonstra esses elementos (h1 a h6).

```

1 <?xml version = "1.0"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5 <!-- Figura J.2: header.html -->
6 <!-- níveis de título XHTML. -->
7
8 <html xmlns = "http://www.w3.org/1999/xhtml">
9 <head>
10 <title>XHTML headers</title>
11 </head>
12
13 <body>
14
15 <h1>Level 1 Header</h1>
16 <h2>Level 2 header</h2>
17 <h3>Level 3 header</h3>
18 <h4>Level 4 header</h4>
19 <h5>Level 5 header</h5>
20 <h6>Level 6 header</h6>
21
22 </body>
23 </html>

```



**Figura J.2** Elementos níveis de título de h1 a h6.

O elemento nível de título h1 (linha 15) é considerado o cabeçalho mais significativo e é renderizado em uma fonte maior que a dos outros cinco níveis de título (linhas 16–20). Cada elemento nível de título sucessivo (isto é, h2, h3 etc.) é renderizado em uma fonte menor.



### Dica de portabilidade J.1

*O tamanho do texto utilizado para exibir cada elemento nível de título pode variar significativamente entre navegadores.*



## Observação sobre a aparência e comportamento J.1

Colocar um nível de título na parte superior de cada página XHTML ajuda os visualizadores a entender o propósito de cada página.



## Observação sobre a aparência e comportamento J.2

Utilize níveis de título maiores para enfatizar seções mais importantes de uma página Web.

## J.5 Vinculando

Um dos recursos XHTML mais importantes é o **hyperlink**, que referencia (ou **vincula-se a**) outros recursos, como documentos XHTML e imagens. Em XHTML, tanto o texto como as imagens podem atuar como hyperlinks. Em geral, os navegadores Web sublinham hyperlinks de texto e os pintam de azul por padrão, para que os usuários possam distinguir entre hyperlinks e texto simples. Na Figura J.3, criamos hyperlinks de texto para quatro sites Web diferentes. A linha 17 introduz o tag **<strong>**. Em geral, os navegadores exibem texto marcado com **<strong>** em uma fonte em negrito.

Os links são criados utilizando o **elemento a (âncora)**. A linha 21 define um hyperlink que vincula o texto Deitel com o URL atribuído ao atributo **href**, que especifica a localização de um recurso vinculado, como uma página Web, um arquivo ou um endereço de correio eletrônico. Esse elemento âncora particular vincula-se a uma página Web localizada em <http://www.deitel.com>. Quando um URL não indica um documento específico no site Web, o servidor Web retorna uma página Web padrão. Essa página é freqüentemente chamada `index.html`; entretanto, a maioria dos servidores Web pode ser configurada para utilizar qualquer arquivo como a página Web padrão do site. (Abra <http://www.deitel.com> em uma janela de navegador e <http://www.deitel.com/index.html> em uma segunda janela de navegador para confirmar que elas são idênticas.) Se o servidor Web não puder localizar um documento solicitado, o servidor retorna uma indicação de erro ao navegador Web e este exibe uma mensagem de erro para o usuário.

As âncoras podem vincular-se a endereços de correio eletrônico por meio de um URL **mailto:**. Quando alguém clicar nesse tipo de link ancorado, a maioria dos navegadores carrega o programa de correio eletrônico padrão (por exemplo, o Outlook Express) para permitir que o usuário escreva uma mensagem de correio eletrônico para o endereço vinculado. A Figura J.4 demonstra esse tipo de âncora.

As linhas 16–18 contêm um link de correio eletrônico. A forma de uma âncora de correio eletrônico é `<a href = "mailto:endereçoDeCorreioEletrônico">...</a>`. Nesse caso, criamos um link para o endereço de correio eletrônico [deitel@deitel.com](mailto:deitel@deitel.com).

```

1 <?xml version = "1.0"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5 <!--Figura J.3: links.html -->
6 <!-- Introdução aos hyperlinks. -->
7
8 <html xmlns = "http://www.w3.org/1999/xhtml">
9 <head>
10 <title>Introduction to hyperlinks</title>
11 </head>
12
13 <body>
14
15 <h1>Here are my favorite sites</h1>
16
17 <p>Click a name to go to that page.</p>
18
19 <!-- cria quatro hyperlinks de texto -->
20 <p>
21 Deitel
22 </p>
23
24 <p>
25 Prentice Hall

```

Figura J.3 Estabelecendo links para outras páginas Web.

(continua)

```

26 </p>
27
28 <p>
29 Yahoo!
30 </p>
31
32 <p>
33 USA Today
34 </p>
35
36 </body>
37 </html>

```

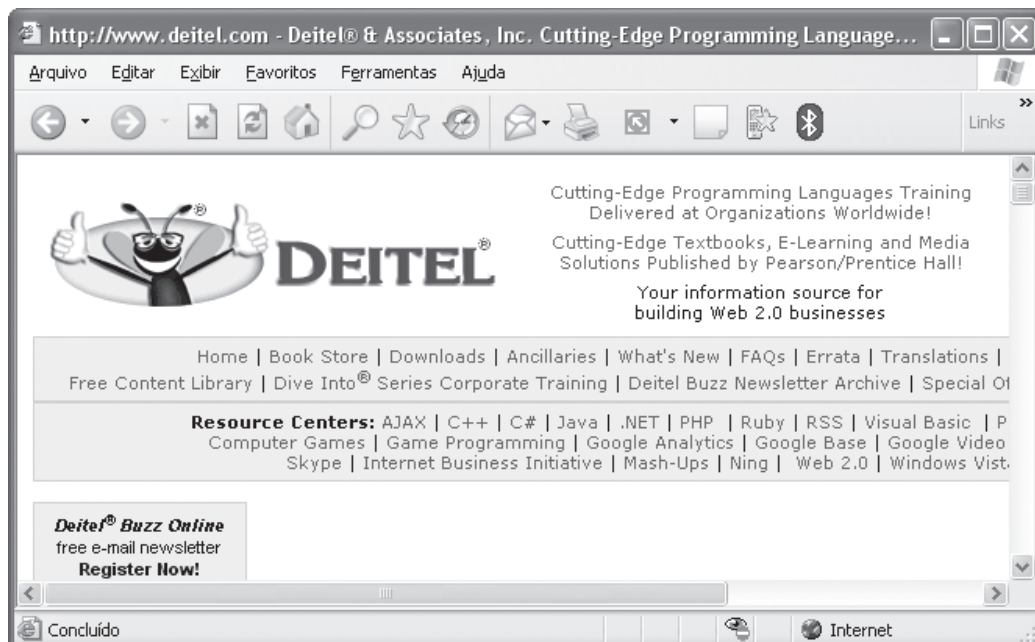
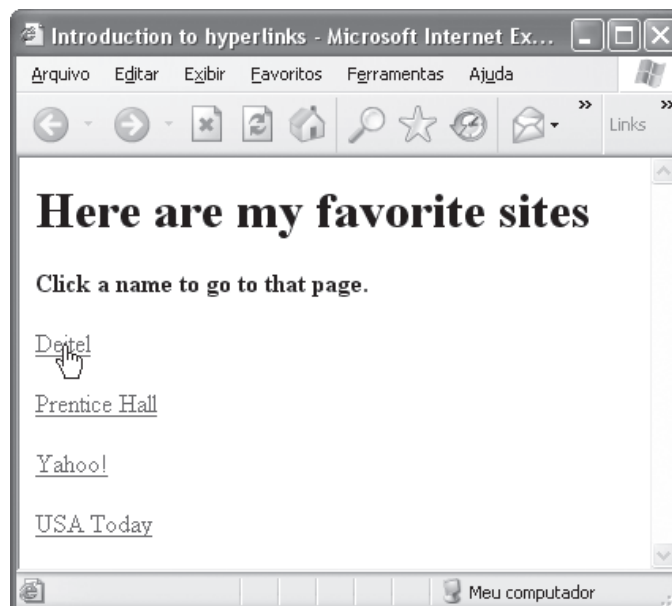


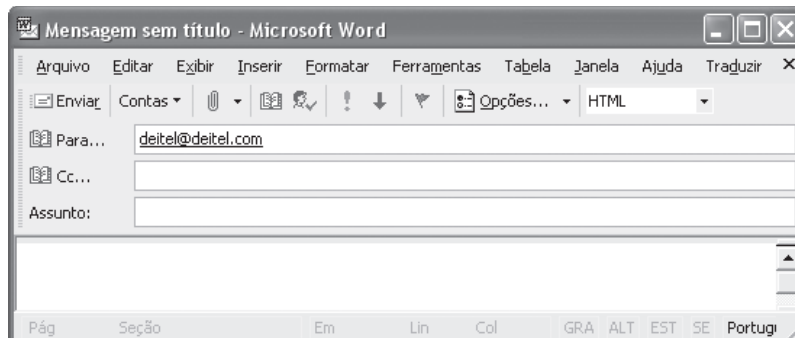
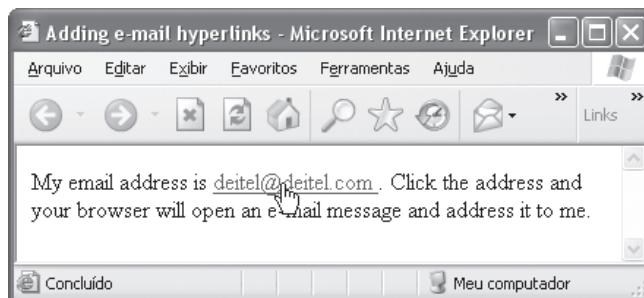
Figura J.3 Estabelecendo links para outras páginas Web.

(continuação)

```

1 <?xml version = "1.0"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5 <!-- Figura J.4: contact.html -->
6 <!-- Adicionando hyperlinks de correio eletrônico. -->
7
8 <html xmlns = "http://www.w3.org/1999/xhtml">
9 <head>
10 <title>Adding e-mail hyperlinks</title>
11 </head>
12
13 <body>
14
15 <p>My email address is
16
17 deitel@deitel.com
18
19 . Click the address and your browser will
20 open an e-mail message and address it to me.
21
22 </p>
23 </body>
24 </html>

```



**Figura J.4** Estabelecendo links para um endereço de correio eletrônico.

## J.6 Imagens

Os exemplos discutidos até agora demonstraram como marcar documentos que só contêm texto. Entretanto, a maioria das páginas Web contém tanto texto como imagens. De fato, as imagens são uma parte igual e essencial do design de página Web. Os dois formatos de imagem mais populares utilizados por desenvolvedores Web são as imagens GIF (Graphics Interchange Format) e JPEG (Joint Photographic Experts Group). Os usuários podem criar imagens, utilizando softwares especializados, como o Adobe® PhotoShop Elements e o Jasc® Paint Shop Pro ([www.jasc.com](http://www.jasc.com)). As imagens também podem ser adquiridas a partir de vários sites Web, como [gallery.yahoo.com](http://gallery.yahoo.com). A Figura J.5 demonstra como incorporar imagens em páginas Web.



## Boa prática de programação J.5

Inclua sempre a *width* (largura) e a *height* (altura) de uma imagem dentro da tag `<img>`. Quando o navegador carregar o arquivo XHTML, ele saberá imediatamente a partir desses atributos quanto espaço de tela oferecer à imagem e organizará a página adequadamente, mesmo antes de ele fazer o download da imagem.



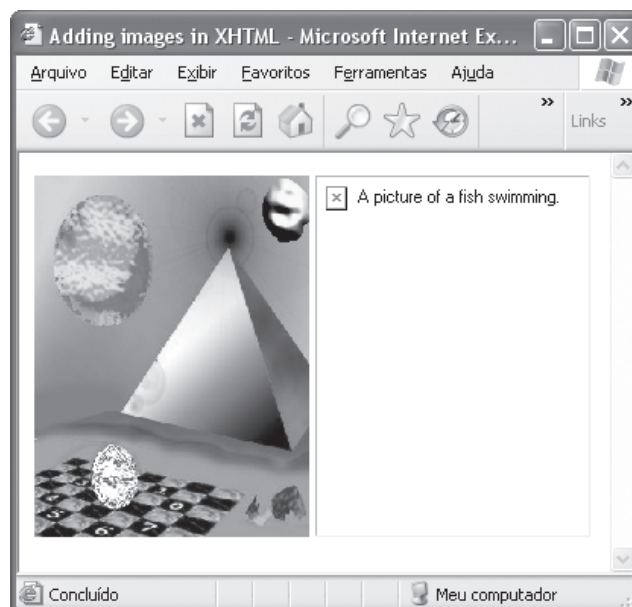
## Dica de desempenho J.1

Incluir os atributos *width* e *height* em um tag `<img>` ajudará o navegador a carregar e renderizar páginas mais rapidamente.

```

1 <?xml version = "1.0"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5 <!-- Figura J.5: picture.html -->
6 <!-- Adicionando imagens com XHTML. -->
7
8 <html xmlns = "http://www.w3.org/1999/xhtml">
9 <head>
10 <title>Adding images in XHTML</title>
11 </head>
12
13 <body>
14
15 <p>
16 <img src = "cool8se.jpg" height = "238" width = "181"
17 alt = "An imaginary landscape." />
18
19 <img src = "fish.jpg" height = "238" width = "181"
20 alt = "A picture of a fish swimming." />
21 </p>
22
23 </body>
24 </html>

```



**Figura J.5** Colocando imagens em arquivos XHTML.



## Erro comum de programação J.4

*Inserir novas dimensões para uma imagem que muda sua relação de largura para altura inerente poderia distorcer a aparência da imagem. Por exemplo, se sua imagem tiver 200 pixels de largura e 100 de altura, você deve assegurar que quaisquer novas dimensões tenham uma relação de altura para largura 2:1.*

As linhas 16–17 utilizam um elemento **img** para inserir uma imagem no documento. A localização do arquivo de imagem é especificada com o atributo **src** do elemento **img**. Nesse caso, a imagem está localizada no mesmo diretório que esse documento XHTML, portanto somente o nome de arquivo da imagem é requerido. Os atributos opcionais **width** e **height** especificam a largura e a altura da imagem, respectivamente. O autor de um documento pode fazer a escala de uma imagem aumentando ou diminuindo os valores dos atributos **width** e **height** da imagem. Se esses atributos são omitidos, o navegador utiliza a largura e a altura reais da imagem. As imagens são medidas em **pixels** (*‘picture elements’*), que representam pontos de cor na tela. A imagem na Figura J.5 tem 181 pixels de largura e 238 pixels de altura.

Cada elemento **img** em um documento XHTML tem um atributo **alt**. Se um navegador não puder renderizar uma imagem, o navegador exibirá o valor do atributo **alt**. Um navegador poderia não ser capaz de renderizar uma imagem por várias razões. Talvez ele não suporte imagens — como é o caso com um **navegador baseado em texto** (isto é, um navegador que exibe somente texto) — ou o cliente pode ter desativado a visualização das imagens para reduzir o tempo de download. A Figura J.5 mostra o Internet Explorer renderizando o valor do atributo **alt** quando um documento referencia um arquivo de imagem inexistente (**fish.jpg**).

O atributo **alt** é importante para criar páginas Web acessíveis para usuários com deficiências, especialmente aqueles com deficiências visuais e navegadores baseados em texto. O software especializado chamado de **sintetizador de fala** é freqüentemente utilizado por portadores de deficiências. Esses aplicativos de software ‘falam’ o valor do atributo **alt** para que o usuário saiba o que o navegador está exibindo.

Alguns elementos XHTML (chamados de **elementos vazios**) contêm somente atributos e não marcam texto (isto é, o texto não é colocado entre os tags inicial e final). Os elementos vazios (por exemplo, **img**) devem ser terminados, utilizando o **caractere de barra normal** (/) dentro do colchete angular de fechamento direito (>) do tag inicial ou incluindo explicitamente o tag final. Ao utilizar o caractere de barra normal, adicionamos um espaço antes da barra normal para melhorar a legibilidade (como mostrado no final das linhas 17 e 20). Em vez de utilizar o caractere de barra normal, as linhas 19–20 poderiam ser escritas com um tag **</img>** de fechamento como mostrado a seguir:

```
<img src = "cool8se.jpg" height = "238" width = "181"
 alt = "An imaginary landscape.">
```

Utilizando imagens como hyperlinks, os desenvolvedores Web podem criar páginas Web gráficas que se vinculam a outros recursos. Na Figura J.6, criamos seis hyperlinks de imagem diferentes.

As linhas 16–19 criam um **hyperlink de imagem** aninhando um elemento **img** dentro de um elemento âncora (**a**). O valor do atributo **src** do elemento **img** especifica que essa imagem (**links.jpg**) reside em um diretório chamado **buttons**. O diretório **buttons** e o documento XHTML estão no mesmo diretório. As imagens de outros documentos Web também podem ser referenciadas (depois de obter a permissão do proprietário do documento) configurando o atributo **src** como o nome e a localização da imagem.

```
1 <?xml version = "1.0"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5 <!-- Figura J.6: nav.html -->
6 <!-- Utilizando imagens como âncoras de link. -->
7
8 <html xmlns = "http://www.w3.org/1999/xhtml">
9 <head>
10 <title>Using images as link anchors</title>
11 </head>
12
13 <body>
14
15 <p>
16
17 <img src = "buttons/links.jpg" width = "65"
18 height = "50" alt = "Links Page" />
19

```

**Figura J.6** Utilizando imagens como âncoras de link.

(continua)



```

20
21
22 <img src = "buttons/list.jpg" width = "65"
23 height = "50" alt = "List Example Page" />
24

25
26
27 <img src = "buttons/contact.jpg" width = "65"
28 height = "50" alt = "Contact Page" />
29

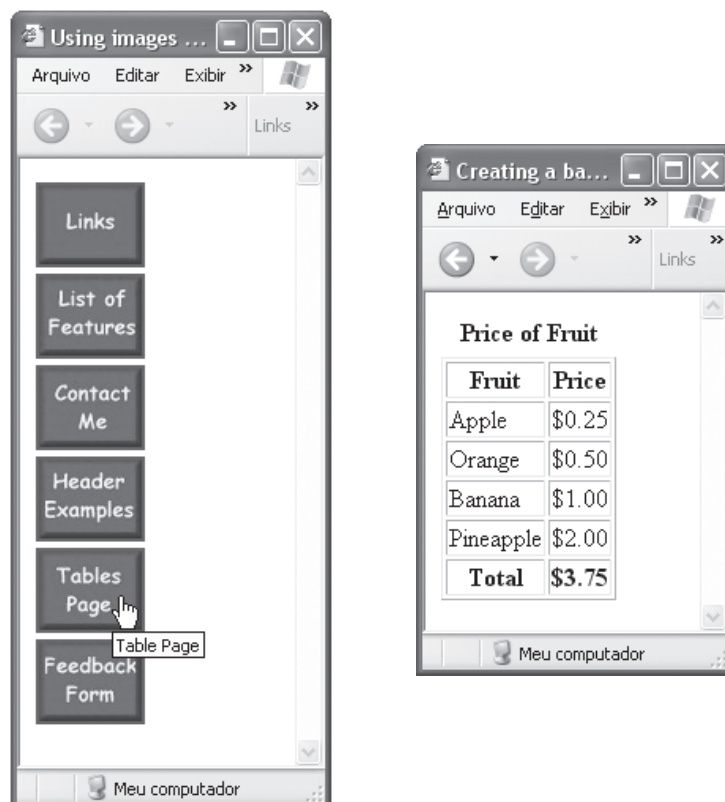
30
31
32 <img src = "buttons/header.jpg" width = "65"
33 height = "50" alt = "Header Page" />
34

35
36
37 <img src = "buttons/table.jpg" width = "65"
38 height = "50" alt = "Table Page" />
39

40
41
42 <img src = "buttons/form.jpg" width = "65"
43 height = "50" alt = "Feedback Form" />
44

45 </p>
46
47 </body>
48 </html>

```



**Figura J.6** Utilizando imagens como âncoras de link.

(continuação)

Na linha 19, introduzimos o **elemento br**, que a maioria de navegadores renderiza como uma **quebra de linha**. Qualquer marcação ou texto que segue um elemento `br` é renderizado na próxima linha. Assim como o elemento `img`, `br` é um exemplo de um elemento vazio terminado com uma barra normal. Adicionamos um espaço antes da barra normal para melhorar a legibilidade.

## J.7 Caracteres especiais e mais quebras de linha

Ao marcar o texto, certos caracteres ou símbolos (por exemplo, `<`) podem ser difíceis de incorporar diretamente em um documento XHTML. Alguns teclados não fornecem esses símbolos, ou a presença desses símbolos poderia causar erro de sintaxe. Por exemplo, a marcação

```
<p>if x < 10 then increment x by 1</p>
```

resulta em um erro de sintaxe, porque utiliza o caractere menor que (`<`), que é reservado para tags iniciais e finais como `<p>` e `</p>`. A XHTML fornece os **caracteres especiais** ou **referências de entidade** (na forma `&código;`) para representar esses caracteres. Poderíamos corrigir a linha anterior escrevendo

```
<p>if x < 10 then increment x by 1</p>
```

que utiliza o caractere especial `&lt;` para o símbolo menor que.

A Figura J.7 demonstra como utilizar caracteres especiais em um documento XHTML. Para uma lista de caracteres especiais, veja o Apêndice K. As linhas 26–27 contêm outros caracteres especiais, que são expressos como abreviações de palavra (por exemplo, `&amp;` para `&`, ou `'e comercial'`, ou *ampersand*, e `&copy;` para `©`, ou marca de direitos autorais) ou como valores hexadecimais (por exemplo, `&#38;`; é a representação hexadecimal de `&`). Os números hexadecimais são números de base 16 — os dígitos em um número hexadecimal têm valores de 0 a 15 (um total de 16 valores diferentes). As letras A–F representam os dígitos hexadecimais correspondentes aos valores decimais 10–15. Portanto, na notação hexadecimal, podemos ter números como 876 consistindo unicamente em dígitos do tipo decimal, números DA19F consistindo em dígitos, e letras e números DCB consistindo unicamente em letras. Discutimos os números hexadecimais em detalhes no Apêndice D.

Nas linhas 33–35, introduzimos três novos elementos. A maioria dos navegadores renderiza o elemento `del` como texto riscado. Com esse formato, os usuários podem indicar facilmente as revisões de documento. Para **sobrescrever** texto (isto é, elevar o texto em uma linha com um tamanho diminuído da fonte) ou **subscrever** o texto (isto é, rebaixar texto em uma linha com um tamanho da fonte diminuído), utilize os elementos `sup` e `sub`, respectivamente. Também utilizamos os caracteres especiais `&lt;` para um sinal menor que e `&frac14;` para o 1/4 de fração (linha 37).

Além dos caracteres especiais, esse documento introduz uma **linha horizontal**, indicada pelo tag `<hr />` na linha 24. A maioria dos navegadores renderiza uma linha horizontal como uma linha horizontal. O tag `<hr />` também insere uma quebra de linha acima e abaixo da linha horizontal.

```

1 <?xml version = "1.0"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5 <!-- Figura J.7: contact2.html -->
6 <!-- Inserindo caracteres especiais. -->
7
8 <html xmlns = "http://www.w3.org/1999/xhtml">
9 <head>
10 <title>Inserting special characters</title>
11 </head>
12
13 <body>
14
15 <!-- caracteres especiais são -->
16 <!-- inseridos usando a forma &code; -->
17 <p>
18 Click
19 here
20 to open an e-mail message addressed to
21 deitel@deitel.com.
22 </p>

```

**Figura J.7** Inserindo caracteres especiais em XHTML.

(continua)

```

23
24 <hr /> <!-- insere uma linha horizontal -->
25
26 <p>All information on this site is ©
27 Deitel & Associates, Inc. 2005.</p>
28
29 <!-- para texto riscado utilize os tags -->
30 <!-- para texto subscrito utilize os tags <sub> -->
31 <!-- para texto sobrescrito utilize os tags <sup> -->
32 <!-- esses tags são aninhados dentro de outros tags -->
33 <p>You may download 3.14 x 10²
34 characters worth of information from this site.
35 Only _{one} download per hour is permitted.</p>
36
37 <p>Note: < ¼ of the information
38 presented here is updated daily.</p>
39
40 </body>
41 </html>

```

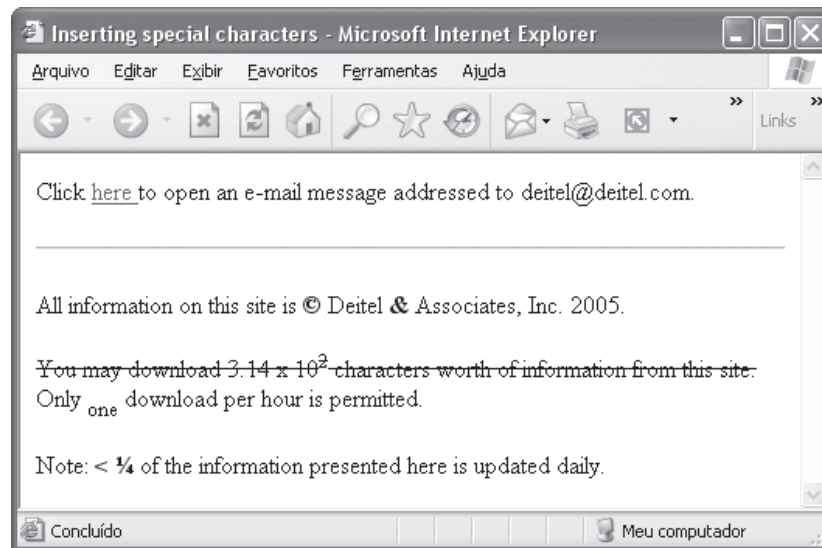


Figura J.7 Inserindo caracteres especiais em XHTML.

(continuação)

## J.8 Listas não ordenadas

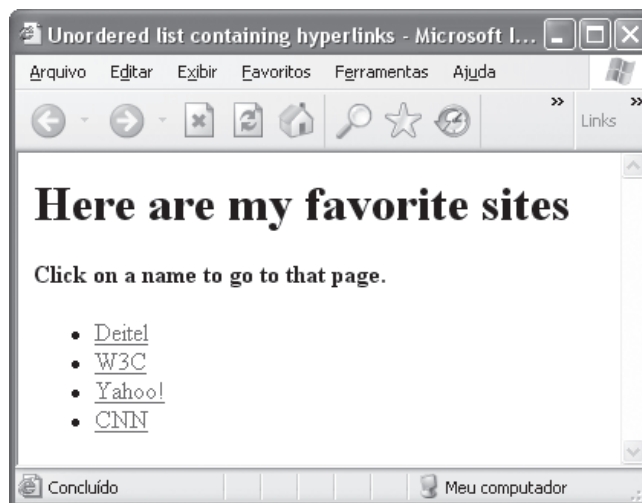
Até esse ponto, apresentamos elementos e atributos XHTML básicos para estabelecer links para recursos, criar cabeçalhos, utilizar caracteres especiais e incorporar imagens. Nesta seção, discutimos como organizar informações sobre uma página Web utilizando listas. Mais adiante no apêndice, introduzimos outro recurso para organizar informações, chamado tabela. A Figura J.8 exibe texto em uma lista não ordenada (isto é, uma lista que não ordena seus itens por letra ou número). O **elemento lista não ordenada u1** cria uma lista em que cada item inicia com um marcador (chamado de **disco**).

Cada entrada em uma lista não ordenada (elemento u1 na linha 20) é um elemento **li** (*list item*) (linhas 23, 25, 27 e 29). A maioria dos navegadores Web renderiza esses elementos com uma quebra de linha e um símbolo de marcador recuado em relação ao início da nova linha.

```

1 <?xml version = "1.0"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5 <!-- Figura J.8: links2.html -->
6 <!-- Lista não ordenada contendo hyperlinks. -->
7
8 <html xmlns = "http://www.w3.org/1999/xhtml">
9 <head>
10 <title>Unordered list containing hyperlinks</title>
11 </head>
12
13 <body>
14
15 <h1>Here are my favorite sites</h1>
16
17 <p>Click on a name to go to that page.</p>
18
19 <!-- cria uma lista não ordenada -->
20
21
22 <!-- adiciona quatro itens de lista -->
23 Deitel
24
25 W3C
26
27 Yahoo!
28
29 CNN
30
31
32
33 </body>
34 </html>

```



**Figura J.8** Listas não ordenadas em XHTML.

## J.9 Listas aninhadas e listas ordenadas

As listas podem ser aninhadas para representar relacionamentos hierárquicos, como em um formato de estrutura de tópicos. A Figura J.9 demonstra listas aninhadas e listas ordenadas (isto é, a lista que ordena itens por letra ou número).

A primeira lista ordenada inicia na linha 33. O atributo **type** especifica o **tipo de seqüência** (isto é, o conjunto de números ou letras utilizado na lista ordenada). Nesse caso, configurar **type** como "I" especifica numerais romanos em letras maiúsculas. A linha 47 inicia a segunda lista ordenada e configura o atributo **type** como "a", especificando letras minúsculas para os itens de lista. A última lista ordenada (linhas 71–75) não utiliza o atributo **type**. Por padrão, os itens da lista são enumerados de um a três.

Um navegador Web recua cada lista aninhada para indicar um relacionamento hierárquico. Por padrão, os itens na lista não ordenada mais externa (linha 18) são precedidos por discos. Os itens de lista aninhados dentro da lista não ordenada da linha 18 são precedidos por círculos. Embora não demonstrados nesse exemplo, os itens de lista aninhados subsequentes são precedidos por quadrados. Os itens de lista não ordenados podem ser configurados explicitamente como discos, círculos ou quadrados configurando o atributo **type** do elemento **ul** como "disc", "circle" ou "square", respectivamente.

## J.10 Tabelas XHTML básicas

Esta seção apresenta a **tabela XHTML** — um recurso frequentemente utilizado que organiza dados em linhas e colunas. Nosso primeiro exemplo (Figura J.10) utiliza uma tabela com seis linhas e duas colunas para exibir informações de preços de frutas.

As tabelas são definidas com o elemento **table**. As linhas 16–18 especificam o tag inicial para um elemento tabela que tem vários atributos. O atributo **border** especifica a largura da borda da tabela em pixels. Para criar uma tabela sem bordas, configure **border** como "0". Esse exemplo atribui o atributo **width** "40%", a fim de configurar a largura da tabela como 40 por cento da largura do navegador. Um desenvolvedor também pode configurar o atributo **width** como um número especificado de pixels.

Como seu nome implica, o atributo **summary** (linha 17) descreve o conteúdo da tabela. Os dispositivos de fala utilizam esse atributo para tornar a tabela mais acessível para usuários com danos visuais. O elemento **caption** (linha 22) descreve o conteúdo da tabela e ajuda navegadores baseados em texto a interpretar os dados da tabela. O texto dentro do tag **<caption>** é exibido acima da tabela pela maioria dos navegadores. O atributo **summary** e o elemento **caption** são dois dos muitos recursos XHTML que tornam as páginas Web mais acessíveis aos usuários com deficiências.



### Dica de prevenção de erro J.1

Tente redimensionar a janela de navegador para ver como a largura da janela afeta a largura da tabela.

```

1 <?xml version = "1.0"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5 <!-- Figura J.9: list.html -->
6 <!-- Listas avançadas: aninhadas e ordenadas. -->
7
8 <html xmlns = "http://www.w3.org/1999/xhtml">
9 <head>
10 <title>Advanced lists</title>
11 </head>
12
13 <body>
14
15 <h1>The Best Features of the Internet</h1>
16
17 <!-- cria uma lista não ordenada -->
18
19 You can meet new people from countries around
20 the world.
21
```

Figura J.9 Listas aninhadas e ordenadas em XHTML.

(continua)

```

22
23 You have access to new media as it becomes public:
24
25 <!-- inicia lista aninhada, utiliza marcadores modificados -->
26 <!-- lista termina com tag de fechamento -->
27
28 New games
29
30 New applications
31
32 <!-- lista aninhada ordenada -->
33 <ol type = "I">
34 For business
35 For pleasure
36
37
38
39
40 Around the clock news
41 Search engines
42 Shopping
43
44 Programming
45
46 <!-- outra lista ordenada aninhada -->
47 <ol type = "a">
48 XML
49 Java
50 XHTML
51 Scripts
52 New languages
53
54
55
56
57 <!-- termina lista aninhada iniciada na linha 27 -->
58
59
60
61 Links
62 Keeping in touch with old friends
63 It is the technology of the future!
64
65 <!-- termina lista não ordenada iniciada na linha 18 -->
66
67 <h1>My 3 Favorite CEOs</h1>
68
69 <!-- elementos ol sem atributo type têm -->
70 <!-- tipo de sequência numérico (isto é, 1, 2, ...) -->
71
72 Lawrence J. Ellison
73 Steve Jobs
74 Michael Dell
75
76
77 </body>
78 </html>

```

Figura J.9 Listas aninhadas e ordenadas em XHTML.

(continua)



**Figura J.9** Listas aninhadas e ordenadas em XHTML.

(continuação)

```

1 <?xml version = "1.0"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5 <!-- Figura J.10: table1.html -->
6 <!-- Criando uma tabela básica. -->
7
8 <html xmlns = "http://www.w3.org/1999/xhtml">
9 <head>
10 <title>Creating a basic table</title>
11 </head>
12
13 <body>
14
```

**Figura J.10** Tabela XHTML.

(continua)



```

15 <!-- o tag <table> inicia a tabela -->
16 <table border = "1" width = "40%"
17 summary = "This table provides information about
18 the price of fruit">
19
20 <!-- o tag <caption> resume o conteúdo da tabela -->
21 <!-- para ajudar deficientes visuais -->
22 <caption>Price of Fruit</caption>
23
24 <!-- <thead> é a primeira seção da tabela -->
25 <!-- ele formata a área de cabeçalho da tabela -->
26 <thead>
27 <tr> <!-- <tr> insere uma linha de tabela -->
28 <th>Fruit</th> <!-- insere célula de título -->
29 <th>Price</th>
30 </tr>
31 </thead>
32
33 <!-- todo conteúdo de tabela é incluído dentro de <tbody> -->
34 <tbody>
35 <tr>
36 <td>Apple</td> <!-- insere célula de dados -->
37 <td>$0.25</td>
38 </tr>
39
40 <tr>
41 <td>Orange</td>
42 <td>$0.50</td>
43 </tr>
44
45 <tr>
46 <td>Banana</td>
47 <td>$1.00</td>
48 </tr>
49
50 <tr>
51 <td>Pineapple</td>
52 <td>$2.00</td>
53 </tr>
54 </tbody>
55
56 <!-- <tfoot> é a última seção da tabela -->
57 <!-- formata rodapé da tabela -->
58 <tfoot>
59 <tr>
60 <th>Total</th>
61 <th>$3.75</th>
62 </tr>
63 </tfoot>
64
65 </table>
66
67 </body>
68 </html>

```

Figura J.10 Tabela XHTML.

(continua)

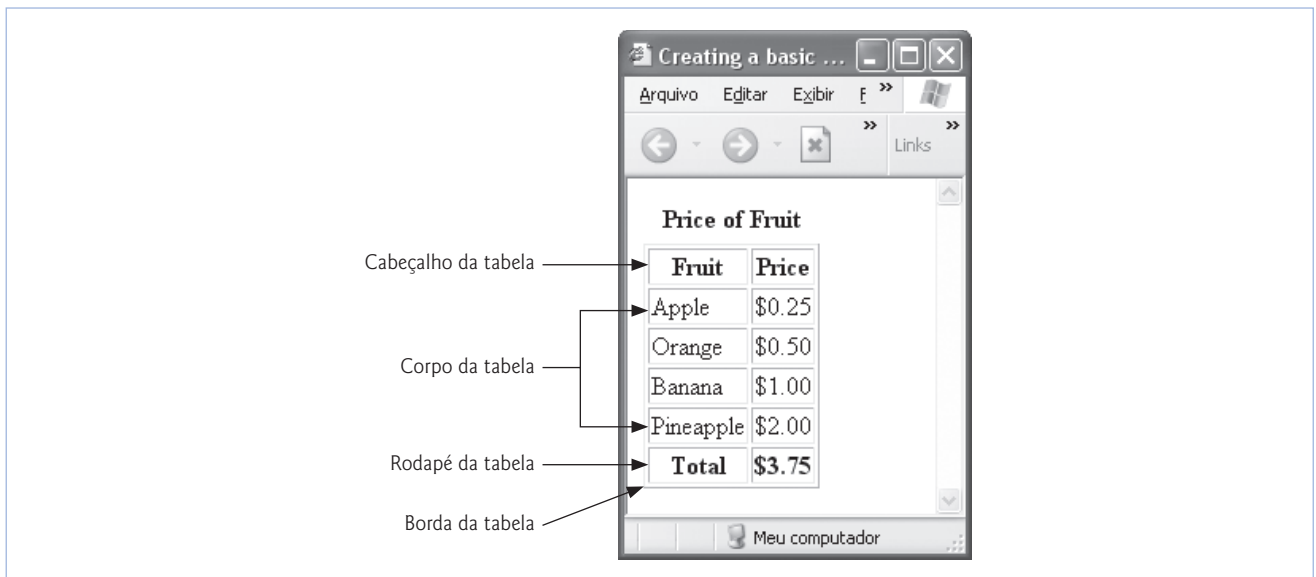


Figura J.10 Tabela XHTML.

(continuação)

Uma tabela tem três seções distintas — **cabeçalho**, **corpo** e **rodapé**. A seção cabeçalho (ou **célula de cabeçalho**) é definida com um elemento **thead** (linhas 26–31), que contém informações de cabeçalho, como nomes de coluna. Cada elemento **tr** (linhas 27–30) define uma **table row** (linha de tabela) individual. As colunas na seção cabeçalho são definidas com elementos **th**. A maioria dos navegadores centraliza texto formatado por elementos **th** (coluna de cabeçalho de tabela) e aparece em negrito. Os elementos cabeçalho de tabela são aninhados dentro de elementos linha de tabela.

A seção corpo, ou **corpo da tabela**, contém os principais dados da tabela. O corpo da tabela (linhas 34–54) é definido em um elemento **tbody**. As **células de dados** contêm partes individuais de dados e são definidas com elementos **td** (**table data**, **dados de tabela**).

A seção rodapé (linhas 58–63) é definida com um elemento **tfoot** (**rodapé de tabela**) e representa um rodapé. O texto comumente colocado no rodapé inclui resultados de cálculo e notas de rodapé. Como outras seções, a seção rodapé pode conter linhas de tabela e cada linha pode conter colunas.

## J.11 Tabelas e formatação XHTML intermediárias

Na seção anterior, exploramos a estrutura de uma tabela básica. Na Figura J.11, aprimoramos nossa discussão de tabelas introduzindo elementos e atributos que permitem ao autor do documento construir tabelas mais complexas.



### Erro comum de programação J.5

*Ao utilizar **colspan** e **rowspan** para ajustar o tamanho das células de dados de tabela, tenha em mente que as células modificadas ocuparão mais de uma coluna ou linha; outras linhas ou colunas da tabela devem compensar as linhas ou colunas extras ocupadas por células individuais. Caso contrário, a formatação da tabela será distorcida e você poderia criar inadvertidamente mais colunas e linhas do que pretendia originalmente.*

A tabela inicia na linha 17. O elemento **colgroup** (linhas 22–27) agrupa e formata colunas. O elemento **col** (linha 26) especifica dois atributos nesse exemplo. O atributo **align** determina o alinhamento do texto na coluna. O atributo **span** determina quantas colunas o elemento **col** formata. Nesse caso, configuramos o valor de **align** como "right" e o valor de **span** como "1" para o texto alinhado à direita na primeira coluna (a coluna contendo a imagem do camelo na captura de tela de exemplo).

As células de tabela são dimensionadas para ajustar os dados que elas contêm. Os autores de documento podem criar células maiores de dados utilizando atributos **rowspan** e **colspan**. Os valores atribuídos a esses atributos especificam o número de linhas ou colunas ocupado por uma célula. O elemento **th** nas linhas 36–39 utiliza o atributo **rowspan** = "2" para permitir à célula contendo a figura do camelo utilizar duas células verticalmente adjacentes (assim a célula se *estende* [spans] por duas linhas). O elemento **th** nas linhas 42–45 utiliza o atributo **colspan** = "4" para alargar a célula de cabeçalho (contendo Camelid comparison e Approximate as of 9/2002) para se estender por quatro células.

A linha 42 introduz o atributo **valign**, que alinha dados verticalmente e pode receber um de quatro valores — "top" alinha dados com a parte superior da célula, "middle" centraliza os dados verticalmente (o padrão para todos os dados e células de cabeçalho), "bottom" alinha dados com a parte inferior da célula e "baseline" ignora as fontes utilizadas para os dados de linha e configura a parte inferior de todo o texto na linha sobre uma **linha de base** comum (isto é, uma linha horizontal com a qual cada caractere é alinhado).

```

1 <?xml version = "1.0"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5 <!-- Figura J.11: table2.html -->
6 <!-- Design intermediário da tabela. -->
7
8 <html xmlns = "http://www.w3.org/1999/xhtml">
9 <head>
10 <title>Intermediate table design</title>
11 </head>
12
13 <body>
14
15 <h1>Table Example Page</h1>
16
17 <table border = "1">
18 <caption>Here is a more complex sample table.</caption>
19
20 <!-- os tags <colgroup> e <col> são -->
21 <!-- utilizados para formatar colunas inteiras -->
22 <colgroup>
23
24 <!-- o atributo span determina quantas -->
25 <!-- colunas o tag <col> afeta -->
26 <col align = "right" span = "1" />
27 </colgroup>
28
29 <thead>
30
31 <!-- rowspans e colspans mesclam o número especificado -->
32 <!-- de células vertical ou horizontalmente -->
33 <tr>
34
35 <!-- mescla duas linhas -->
36 <th rowspan = "2">
37 <img src = "camel.gif" width = "205"
38 height = "167" alt = "Picture of a camel" />
39 </th>
40
41 <!-- mescla quatro colunas -->
42 <th colspan = "4" valign = "top">
43 <h1>Camelid comparison</h1>

44 <p>Approximate as of 9/2002</p>
45 </th>
46 </tr>
47
48 <tr valign = "bottom">
49 <th># of Humps</th>
50 <th>Indigenous region</th>
51 <th>Spits?</th>
52 <th>Produces Wool?</th>
53 </tr>
54
55 </thead>
56

```

Figura J.11 Tabela XHTML complexa.

(continua)

```

57 <tbody>
58
59 <tr>
60 <th>Camels (bactrian)</th>
61 <td>2</td>
62 <td>Africa/Asia</td>
63 <td rowspan = "2">Llama</td>
64 <td rowspan = "2">Llama</td>
65 </tr>
66
67 <tr>
68 <th>Llamas</th>
69 <td>1</td>
70 <td>Andes Mountains</td>
71 </tr>
72
73 </tbody>
74
75 </table>
76
77 </body>
78 </html>

```

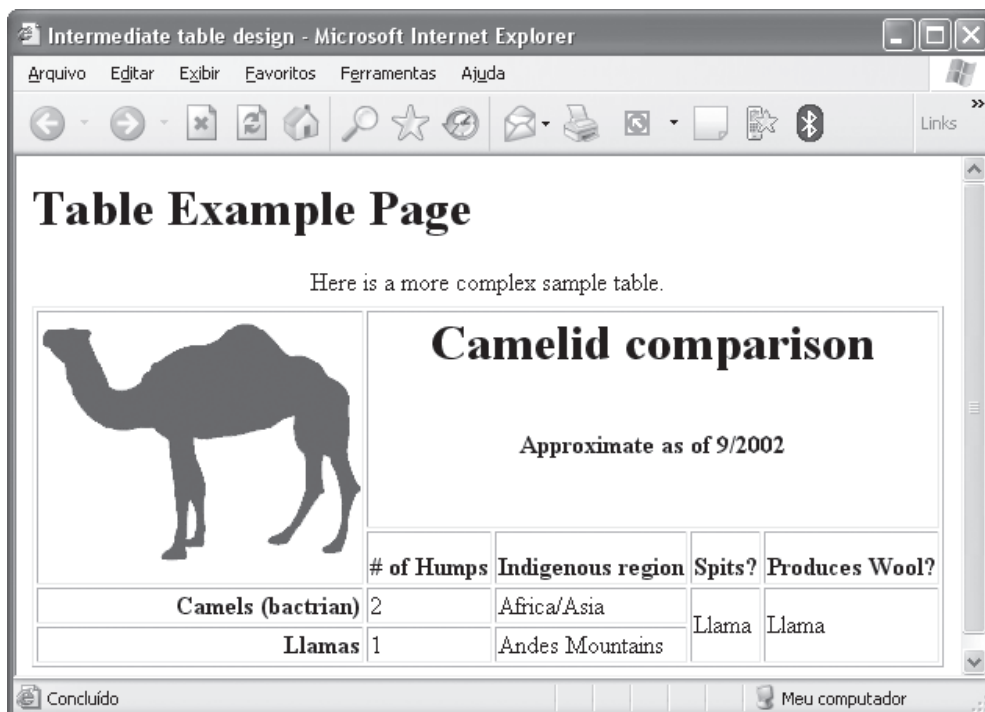


Figura J.11 Tabela XHTML complexa.

(continuação)

## J.12 Formulários XHTML básicos

Ao navegar por sites Web, os usuários muitas vezes precisam fornecer informações como endereços de correio eletrônico, palavras-chave de pesquisa e CEP. A XHTML fornece um mecanismo, chamado **formulário**, para coletar essas informações sobre o usuário.

Os dados que os usuários inserem em uma página Web normalmente são enviados para um servidor Web que fornece acesso a recursos de um site (por exemplo, documentos ou imagens XHTML). Esses recursos estão localizados na mesma máquina que o servidor Web ou em uma máquina que o servidor Web pode acessar pela rede. Quando um navegador solicita uma página Web ou um arquivo que está localizado em um servidor, o servidor processa a solicitação e retorna o recurso solicitado. Uma solicitação contém o nome e

caminho do recurso desejado e o método de comunicação (chamado **protocolo**). Os documentos XHTML utilizam o HTTP (Hypertext Transfer Protocol).

A Figura J.12 envia os dados de formulário para o servidor Web, que os passa para um script CGI (*Common Gateway Interface*) (isto é, um programa) escrito em C++, C, Perl ou alguma outra linguagem. O script processa os dados recebidos do servidor Web e, em geral, retorna as informações para o servidor Web. O servidor Web então envia as informações na forma de um documento XHTML para o navegador Web. [Nota: Este exemplo demonstra funcionalidade do lado do cliente. Se o formulário for enviado (clicando em Submit Your Entries), ocorre um erro.]

Os formulários podem conter componentes visuais e não visuais. Os componentes visuais incluem botões clicáveis e outros componentes de interface gráfica com o usuário com os quais os usuários interagem. Os componentes não visuais, chamados de **entradas ocultas**, armazenam quaisquer dados que o autor do documento especifica, como endereços de correio eletrônico e nomes de arquivo de documento XHTML que atuam como links. O formulário inicia na linha 23 com o elemento `form`. O atributo `method` especifica como os dados do formulário são enviados para o servidor Web.

Utilizar `method = "post"` acrescenta os dados do formulário à solicitação do navegador, que contém o protocolo (isto é, HTTP) e o URL do recurso solicitado. Os scripts localizados no computador do servidor Web (ou em um computador acessível pela rede) podem acessar os dados do formulário enviados como parte da solicitação. Por exemplo, um script pode aceitar as informações do formulário e atualizar uma lista de correio eletrônico. O outro possível valor, `method = "get"`, acrescenta os dados do formulário diretamente ao fim do URL. Por exemplo, o URL `/cgi-bin/formmail` poderia ter as informações de formulário `name = bob` acrescentadas a ele.

O atributo **action** na tag `<form>` especifica o URL de um script no servidor Web; nesse caso, ele especifica um script que envia por e-mail os dados de formulário para um endereço. A maioria dos provedores de serviço da Internet (*Internet service provider* – ISP) tem um script como esse em seu site; pergunte ao administrador de sistema de site Web como configurar um documento XHTML para utilizar o script corretamente.

As linhas 29–36 definem três elementos `input` que especificam dados para fornecer ao script que processa o formulário (também chamado de **handler de formulário**). Esses três elementos `input` têm o atributo `type = "hidden"`, que permite ao autor do documento enviar dados de formulário que não são inseridos em um script por um usuário.

As três entradas ocultas são um endereço de correio eletrônico para o qual os dados serão enviados, a linha de assunto do correio eletrônico e um URL em que o navegador será redirecionado depois de enviar o formulário. Dois outros atributos `input` são **name**, que identifica o elemento `input` e **value**, que fornece o valor que será enviado (ou postado) para o servidor Web.



## Boa prática de programação J.6

*Coloque elementos `input` ocultos no começo de um formulário, imediatamente depois do tag `<form>` de abertura. Essa colocação permite aos autores do documento localizar elementos `input` ocultos rapidamente.*

```

1 <?xml version = "1.0"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5 <!-- Figura J.12: form.html -->
6 <!-- Exemplo 1 de design de formulário. -->
7
8 <html xmlns = "http://www.w3.org/1999/xhtml">
9 <head>
10 <title>Form design example 1</title>
11 </head>
12
13 <body>
14
15 <h1>Feedback Form</h1>
16
17 <p>Please fill out this form to help
18 us improve our site.</p>
19
20 <!-- o tag <form> inicia o formulário, fornece -->
21 <!-- o método de envio de informações -->
22 <!-- e a localização de scripts de formulário -->
23 <form method = "post" action = "/cgi-bin/formmail">
```

**Figura J.12** Formulário simples com campos ocultos e uma caixa de texto.

(continua)

```

24
25 <p>
26
27 <!-- entradas ocultas contêm -->
28 <!-- informações não visuais -->
29 <input type = "hidden" name = "recipient"
30 value = "deitel@deitel.com" />
31
32 <input type = "hidden" name = "subject"
33 value = "Feedback Form" />
34
35 <input type = "hidden" name = "redirect"
36 value = "main.html" />
37 </p>
38
39 <!-- <input type = "text"> insere caixa de texto -->
40 <p>
41 <label>Name:
42 <input name = "name" type = "text" size = "25"
43 maxlength = "30" />
44 </label>
45 </p>
46
47 <p>
48
49 <!-- os input types "submit" e "reset" -->
50 <!-- inserem botão para enviar -->
51 <!-- e limpar o conteúdo do formulário -->
52 <input type = "submit" value =
53 "Submit Your Entries" />
54
55 <input type = "reset" value =
56 "Clear Your Entries" />
57 </p>
58
59 </form>
60
61 </body>
62 </html>

```

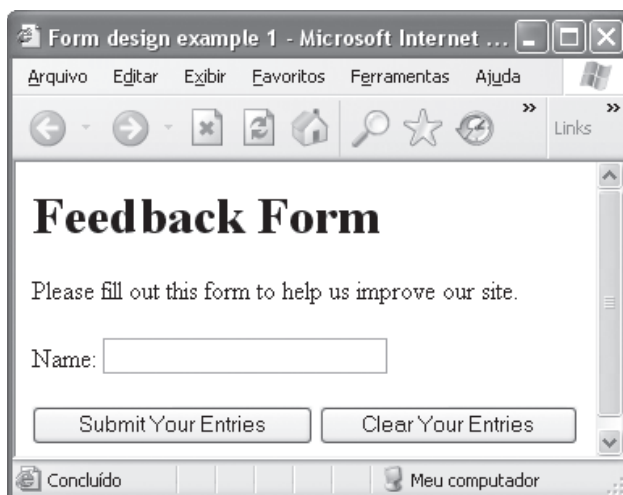


Figura J.12 Formulário simples com campos ocultos e uma caixa de texto.

(continuação)

Introduzimos outro type de input na linha 39. O "text" input insere uma **caixa de texto** no formulário. Os usuários podem digitar dados em caixas de texto. O elemento **rótulo** (linhas 41–44) fornece aos usuários as informações sobre o propósito do elemento input.



### Erro comum de programação J.6

*Esquecer de incluir um elemento label a cada elemento de formulário é um erro de design. Sem esses rótulos, os usuários não podem determinar o propósito de elementos de formulário de indivíduo.*

O atributo **size** do elemento input especifica o número de caracteres visíveis na caixa de texto. O atributo opcional **maxlength** limita o número de entrada de caracteres na caixa de texto. Nesse caso, o usuário não tem permissão de digitar mais que 30 caracteres na caixa de texto.

Há dois tipos de elementos input nas linhas 52–56. O elemento "submit" input é um botão. Quando o usuário pressiona um botão "submit", o navegador envia os dados do formulário ao servidor Web para processamento. O atributo **value** configura o texto exibido no botão (o valor-padrão é Submit). O elemento "reset" input permite a um usuário redefinir todos os elementos form para seus valores-padrão. O atributo **value** do elemento "reset" input configura o texto exibido no botão (o valor-padrão é Reset).

## J.13 Formulários XHTML mais complexos

Na seção anterior, introduzimos os formulários básicos. Nesta seção, introduzimos elementos e atributos para criar formulários mais complexos. A Figura J.13 contém um formulário que solicita feedback de usuário sobre um site Web.

O elemento **textarea** (linhas 42–44) insere uma caixa de texto de múltiplas linhas, chamado de **área de texto**, no formulário. O número de linhas é especificado com o **atributo rows**; e o número de colunas (isto é, caracteres) é especificado com o **atributo cols**. Nesse exemplo, textarea tem quatro linhas de altura e 36 caracteres de largura. Para exibir o texto-padrão na área de texto, coloque o texto entre os tags <textarea> e </textarea>. O texto-padrão pode ser especificado em outros tipos input, como caixas de texto, utilizando o atributo **value**.

O **input type "password"** nas linhas 52–53 insere uma caixa de senha com o **size** especificado. Uma caixa de senha permite aos usuários inserir informações sigilosas, como números de cartão de crédito e senhas, 'mascarando' a entrada de informações com asteriscos. O valor da entrada real é enviado para o servidor Web, não os asteriscos que mascaram a entrada.

As linhas 60–78 introduzem o elemento form **checkbox (caixa de seleção)**. As caixas de seleção permitem aos usuários selecionar opções a partir de um conjunto. Quando um usuário seleciona uma caixa de seleção, uma marca aparece nessa caixa. Caso contrário, a caixa de seleção permanece vazia. Cada "checkbox" input cria uma nova caixa de seleção. As caixas de seleção podem ser utilizadas individualmente ou em grupos. As caixas de seleção que pertencem a um grupo recebem o mesmo nome (nesse caso, "thingsIiked").

Continuamos nossa discussão de formulários apresentando um terceiro exemplo que introduz mais elementos de formulário a partir dos quais usuários podem fazer seleções (Figura J.14). Neste exemplo, introduzimos dois novos tipos de inputs. O primeiro tipo é o **botão de opção** (linhas 90–113), especificado com o tipo "radio". Os botões de opção são semelhantes às caixas de seleção, exceto pelo fato de que apenas um botão de opção em um grupo de botões de opção pode ser selecionado por vez. Todos os botões de opção em um grupo têm o mesmo atributo **name**; são distinguidos por seus atributos **value** diferentes. O par atributo–valor **checked = "checked"** (linha 92) indica que botão de opção, se houver algum, é selecionado inicialmente. O atributo **checked** também se aplica a caixas de seleção.



### Erro comum de programação J.7

*Quando seu form tiver várias caixas de seleção com o mesmo name, você deve certificar-se de que eles têm diferentes values ou os scripts que executam no servidor Web não serão capazes de distinguir entre eles.*

```

1 <?xml version = "1.0"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5 <!-- Figura J.13: form2.html -->
6 <!-- Exemplo 2 de design de formulário. -->
7
8 <html xmlns = "http://www.w3.org/1999/xhtml">
9 <head>
10 <title>Form design example 2</title>

```

Figura J.13 Formulário com áreas de texto, caixas de senha e caixas de seleção.

(continua)



```

11 </head>
12
13 <body>
14
15 <h1>Feedback Form</h1>
16
17 <p>Please fill out this form to help
18 us improve our site.</p>
19
20 <form method = "post" action = "/cgi-bin/formmail">
21
22 <p>
23 <input type = "hidden" name = "recipient"
24 value = "deitel@deitel.com" />
25
26 <input type = "hidden" name = "subject"
27 value = "Feedback Form" />
28
29 <input type = "hidden" name = "redirect"
30 value = "main.html" />
31 </p>
32
33 <p>
34 <label>Name:
35 <input name = "name" type = "text" size = "25" />
36 </label>
37 </p>
38
39 <!-- <textarea> cria caixa de texto multilinha -->
40 <p>
41 <label>Comments:

42 <textarea name = "comments" rows = "4"
43 cols = "36">Enter your comments here.
44 </textarea>
45 </label></p>
46
47 <!-- <input type = "password"> insere -->
48 <!-- caixa de texto cuja exibição é mascarada -->
49 <!-- com caracteres de asterisco -->
50 <p>
51 <label>E-mail Address:
52 <input name = "email" type = "password"
53 size = "25" />
54 </label>
55 </p>
56
57 <p>
58 Things you liked:

59
60 <label>Site design
61 <input name = "thingsliked" type = "checkbox"
62 value = "Design" /></label>
63
64 <label>Links
65 <input name = "thingsliked" type = "checkbox"
66 value = "Links" /></label>

```

**Figura J.13** Formulário com áreas de texto, caixas de senha e caixas de seleção.

(continua)

```

67
68 <label>Ease of use
69 <input name = "thingsliked" type = "checkbox"
70 value = "Ease" /></label>
71
72 <label>Images
73 <input name = "thingsliked" type = "checkbox"
74 value = "Images" /></label>
75
76 <label>Source code
77 <input name = "thingsliked" type = "checkbox"
78 value = "Code" /></label>
79 </p>
80
81 <p>
82 <input type = "submit" value =
83 "Submit Your Entries" />
84
85 <input type = "reset" value =
86 "Clear Your Entries" />
87 </p>
88
89 </form>
90
91 </body>
92 </html>

```

**Figura J.13** Formulário com áreas de texto, caixas de senha e caixas de seleção.

(continuação)



## Erro comum de programação J.8

*Ao utilizar um grupo de botões de opção em um formulário, esquecer de configurar os atributos name com o mesmo nome é um erro de lógica que deixa o usuário selecionar todos os botões de opção ao mesmo tempo.*

O elemento **select** (linhas 123–136) fornece uma lista drop-down a partir da qual o usuário pode selecionar um item. O atributo **name** identifica a lista drop-down. O elemento **option** (linhas 124–135) adiciona itens à lista drop-down. O atributo **selected** do elemento **option** especifica que item é inicialmente exibido como o item selecionado no elemento **select**.

```

1 <?xml version = "1.0"?>
2 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
3 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
4
5 <!-- Figura J.14: form3.html -->
6 <!-- Exemplo 3 de design de formulário. -->
7
8 <html xmlns = "http://www.w3.org/1999/xhtml">
9 <head>
10 <title>Form design example 3</title>
11 </head>
12
13 <body>
14
15 <h1>Feedback Form</h1>
16
17 <p>Please fill out this form to help
18 us improve our site.</p>
19
20 <form method = "post" action = "/cgi-bin/formmail">
21
22 <p>
23 <input type = "hidden" name = "recipient"
24 value = "deitel@deitel.com" />
25
26 <input type = "hidden" name = "subject"
27 value = "Feedback Form" />
28
29 <input type = "hidden" name = "redirect"
30 value = "main.html" />
31 </p>
32
33 <p>
34 <label>Name:
35 <input name = "name" type = "text" size = "25" />
36 </label>
37 </p>
38
39 <p>
40 <label>Comments:

41 <textarea name = "comments" rows = "4"
42 cols = "36"></textarea>
43 </label>
44 </p>
45

```

**Figura J.14** O formulário que inclui botões de opção e listas drop-down.

(continua)

```

46 <p>
47 <label>E-mail Address:
48 <input name = "email" type = "password"
49 size = "25" />
50 </label>
51 </p>
52
53 <p>
54 Things you liked:

55
56 <label>Site design
57 <input name = "thingsliked" type = "checkbox"
58 value = "Design" />
59 </label>
60
61 <label>Links
62 <input name = "thingsliked" type = "checkbox"
63 value = "Links" />
64 </label>
65
66 <label>Ease of use
67 <input name = "thingsliked" type = "checkbox"
68 value = "Ease" />
69 </label>
70
71 <label>Images
72 <input name = "thingsliked" type = "checkbox"
73 value = "Images" />
74 </label>
75
76 <label>Source code
77 <input name = "thingsliked" type = "checkbox"
78 value = "Code" />
79 </label>
80
81 </p>
82
83 <!-- <input type = "radio" /> cria um botão de -->
84 <!-- opção. A diferença entre botões de opção -->
85 <!-- e caixas de seleção é que apenas um botão -->
86 <!-- de opção em um grupo pode ser selecionado. -->
87 <p>
88 How did you get to our site?:

89
90 <label>Search engine
91 <input name = "howtosite" type = "radio"
92 value = "search engine" checked = "checked" />
93 </label>
94
95 <label>Links from another site
96 <input name = "howtosite" type = "radio"
97 value = "link" />
98 </label>
99

```

**Figura J.14** O formulário que inclui botões de opção e listas drop-down.

(continua)

```

100 <label>Deitel.com Web site
101 <input name = "howtosite" type = "radio"
102 value = "deitel.com" />
103 </label>
104
105 <label>Reference in a book
106 <input name = "howtosite" type = "radio"
107 value = "book" />
108 </label>
109
110 <label>Other
111 <input name = "howtosite" type = "radio"
112 value = "other" />
113 </label>
114
115 </p>
116
117 <p>
118 <label>Rate our site:
119
120 <!-- o tag <select> apresenta uma lista drop-down -->
121 <!-- com escolhas indicadas pelos -->
122 <!-- tags <option> -->
123 <select name = "rating">
124 <option selected = "selected">Amazing</option>
125 <option>10</option>
126 <option>9</option>
127 <option>8</option>
128 <option>7</option>
129 <option>6</option>
130 <option>5</option>
131 <option>4</option>
132 <option>3</option>
133 <option>2</option>
134 <option>1</option>
135 <option>Awful</option>
136 </select>
137
138 </label>
139 </p>
140
141 <p>
142 <input type = "submit" value =
143 "Submit Your Entries" />
144
145 <input type = "reset" value = "Clear Your Entries" />
146 </p>
147
148 </form>
149
150 </body>
151 </html>

```

**Figura J.14** O formulário que inclui botões de opção e listas drop-down.

(continua)

**Form design example 3 - Microsoft Internet Explorer**

Arquivo Editar Exibir Favoritos Ferramentas Ajuda

Links

## Feedback Form

Please fill out this form to help us improve our site.

Name:

Comments:

E-mail Address:

**Things you liked:**

Site design ☐ Links ☐ Ease of use ☐ Images ☐ Source code ☐

**How did you get to our site?:**

Search engine ☒ Links from another site ☐ Deitel.com Web site ☐ Reference in a book ☐ Other ☐

Rate our site:

Concluído Meu computador

**Figura J.14** O formulário que inclui botões de opção e listas drop-down.

(continuação)

## J.14 Recursos na Internet e na Web

[www.w3.org/TR/xhtml11](http://www.w3.org/TR/xhtml11)

A *XHTML 1.1 Recommendation* contém informações gerais, informações sobre questões de compatibilidade, informações de definição de tipo de documento, definições, terminologia e muito mais informações relacionadas ao XHTML.

[www.xhtml.org](http://www.xhtml.org)

O *XHTML.org* fornece notícias sobre desenvolvimento em XHTML e links para outros recursos XHTML, que incluem livros e artigos.

[www.w3schools.com/xhtml/default.asp](http://www.w3schools.com/xhtml/default.asp)

A *XHTML School* fornece questionários e referências XHTML. Esta página também contém links para a sintaxe XHTML, validação e definições de tipo de documento.

[hotwired.lycos.com/webmonkey/00/50/index2a.html](http://hotwired.lycos.com/webmonkey/00/50/index2a.html)

Este site fornece um artigo sobre XHTML. Seções-chave do artigo fornecem uma visão geral da XHTML e discutem tags, atributos e âncoras.

[wdvl.com/Authoring/Languages/XML/XHTML](http://wdvl.com/Authoring/Languages/XML/XHTML)

A Web Developers' Virtual Library fornece uma introdução à XHTML. Este site também contém artigos, exemplos e links para outras tecnologias.

## Resumo

- A XHTML (Extensible Hypertext Markup Language) é uma linguagem de marcação para criar páginas Web.
- Uma questão-chave ao utilizar XHTML é a separação da apresentação de um documento (isto é, a aparência do documento quando renderizado por um navegador) da estrutura das informações no documento.
- Em XHTML, o texto é marcado com elementos, delimitado por tags que são nomes contidos entre pares de colchetes angulares. Alguns elementos podem conter marcação adicional chamada de atributos, que fornecem informações adicionais sobre o elemento.
- Uma máquina que executa a parte especializada do software, chamada de servidor Web, armazena documentos XHTML.
- Documentos XHTML que estão sintaticamente corretos seguramente renderizam de maneira apropriada. Os documentos XHTML que contêm erro de sintaxe podem não ser exibidos de maneira apropriada.
- Cada documento XHTML contém um tag `<html>` inicial e um tag `</html>` final.
- Os comentários em XHTML sempre iniciam com `<!--` e terminam com `-->`. O navegador ignora todo o texto dentro de um comentário.
- Cada documento XHTML contém um elemento `head`, que geralmente contém informações, como um título e um elemento `body`, que contém o conteúdo da página. As informações no elemento `head` geralmente não são renderizadas na janela de exibição, mas poderiam ser disponibilizadas para o usuário por outro meio.
- O elemento `title` nomeia uma página Web. O título normalmente aparece na barra colorida (chamada de barra de título) na parte superior da janela de navegador e também aparece como o texto que identifica uma página quando os usuários adicionam sua página à lista de **Favorites** ou **Bookmarks**.
- O corpo de um documento XHTML é a área em que o conteúdo do documento é colocado. O conteúdo pode incluir texto e tags.
- Todo texto colocado entre os tags `<p>` e `</p>` forma um parágrafo.
- A XHTML fornece seis níveis de título (`h1` a `h6`) para especificar a importância relativa das informações. O elemento de nível de título `h1` é considerado o nível de título mais significativo e é renderizado em uma fonte maior que os outros cinco cabeçalhos. Cada elemento nível de título sucessivo (isto é, `h2`, `h3` etc.) é renderizado em uma fonte menor.
- Os navegadores Web, em geral, sublinham hyperlinks de texto e os pinta de azul por padrão.
- O tag `<strong>` normalmente faz com que um navegador exiba o texto em uma fonte negrito.
- Os usuários podem inserir links com o elemento `a` (âncora). O atributo mais importante para o elemento `a` é `href`, que especifica o recurso (por exemplo, página, arquivo, endereço de correio eletrônico) sendo vinculado.
- As âncoras podem estabelecer um link para um endereço de correio eletrônico utilizando um URL `mailto:`. Quando alguém clica nesse tipo de link ancorado, a maioria dos navegadores carrega o programa de correio eletrônico padrão (por exemplo, o Outlook Express) para começar a enviar as mensagens de correio eletrônico para os endereços vinculados.
- O atributo `src` do elemento `img` especifica a localização de uma imagem. Os atributos opcionais `width` e `height` especificam a largura e a altura da imagem, respectivamente. As imagens são medidas em pixels (*'picture elements'*), que representam pontos de cor na tela.
- O atributo `alt` torna as páginas Web mais acessíveis a usuários com deficiências, especialmente aqueles com deficiências visuais.
- Alguns elementos XHTML são elementos vazios, contêm apenas atributos e não marcam o texto. Os elementos vazios (por exemplo, `img`) devem ser terminados, utilizando o caractere de barra normal (`/`) ou escrevendo explicitamente um tag final.
- O elemento `br` faz com que a maioria dos navegadores renderize uma quebra de linha. Qualquer marcação ou texto que segue um elemento `br` é renderizado na próxima linha.
- A XHTML fornece caracteres especiais ou referências de entidade (na forma `&código;`) para representar caracteres que não podem ser marcados.
- A maioria dos navegadores renderiza uma linha horizontal, indicada pelo tag `<hr />`, como uma linha horizontal. O elemento `hr` também insere uma quebra de linha acima e outra abaixo da linha horizontal.
- O elemento `ul` de lista não ordenada cria uma lista em que cada item na lista inicia com um símbolo de marcador (chamado de disco). Cada entrada em uma lista não ordenada é um elemento `li` (Item de lista). A maioria dos navegadores Web renderiza esses elementos com uma quebra de linha e um símbolo de marcador no começo da linha.
- As listas podem ser aninhadas para representar relacionamentos hierárquicos de dados.
- O atributo `type` especifica o tipo de sequência (isto é, o conjunto de números ou letras utilizadas na lista ordenada).
- As tabelas XHTML marcam os dados tabulares e são uns dos recursos mais frequentemente utilizados em XHTML.
- O elemento `table` define uma tabela XHTML. O atributo `border` especifica a largura da borda da tabela, em pixels. As tabelas sem bordas configuram esse atributo como "0".
- O elemento `summary` resume o conteúdo da tabela e é utilizado por dispositivos de fala para tornar a tabela mais acessível a usuários com deficiências visuais.



- O elemento `caption` descreve o conteúdo da tabela. O texto dentro do tag `<caption>` é renderizado acima da tabela na maioria dos navegadores.
- Uma tabela pode ser dividida em três seções distintas: cabeçalho (`thead`), corpo (`tbody`) e rodapé (`tfoot`). A seção cabeçalho contém informações como títulos de tabela e cabeçalhos de coluna. O corpo da tabela contém os principais dados da tabela. O rodapé de tabela contém informações como notas de rodapé.
- O elemento `tr` (*table row*), ou linha da tabela, define linhas de tabela individuais. O elemento `th` (*table header*) define uma célula de cabeçalho. O texto em elementos `th` normalmente é centralizado e exibido em negrito pela maioria dos navegadores. Esse elemento pode estar presente em qualquer seção da tabela.
- Os dados dentro de uma linha são definidos com elementos `td`, ou dados de tabela (*table data*).
- O elemento `colgroup` agrupa e formata colunas. Cada elemento `col` pode formatar qualquer número de colunas (especificado com o atributo `span`).
- O autor do documento tem a capacidade de mesclar células de dados com os atributos `rowspan` e `colspan`. Os valores atribuídos a esses atributos especificam o número de linhas ou colunas ocupado pela célula. Esses atributos podem ser colocados dentro de qualquer tag de célula de dados.
- A XHTML fornece formulários para coletar informações de usuários. Os formulários contêm componentes visuais, como botões nos quais os usuários clicam. Os formulários também podem conter componentes não visuais, chamados de entradas ocultas, que são utilizadas para armazenar quaisquer dados, como endereços de correio eletrônico e nomes de arquivo de documento XHTML utilizados para estabelecer links.
- Um formulário inicia com o elemento `form`. O atributo `method` especifica como os dados do formulário são enviados para o servidor Web.
- A entrada "text" insere uma caixa de texto no formulário. As caixas de texto permitem ao usuário inserir dados.
- O atributo `size` do elemento `input` especifica o número de caracteres visíveis no elemento `input`. O atributo opcional `maxlength` limita o número de entrada de caracteres em uma caixa de texto.
- A entrada "submit" envia os dados inseridos no formulário ao servidor Web para processamento. A maioria dos navegadores Web cria um botão que envia os dados de formulário quando clicado. A entrada "reset" permite a um usuário redefinir todos os elementos `form` com seus valores-padrão.
- O elemento `textarea` insere uma caixa de texto multilinha, chamada de área de texto, em um formulário. O número de linhas na área de texto é especificado com o atributo `rows`; e o número de colunas (isto é, caracteres) é especificado com o atributo `cols`.
- O `input type="password"` insere uma caixa de senha em um formulário. Uma caixa de senha permite aos usuários inserir informações sensíveis, como números de cartão de crédito e senhas, 'mascarando' a entrada de informações com outro caractere. Os asteriscos são o caractere de máscara utilizado para caixas de senha. O valor da entrada real é enviado para o servidor Web, não os asteriscos que mascaram a entrada.
- O tipo de entrada caixa de seleção permite ao usuário fazer uma seleção. Quando a caixa de seleção é marcada, um sinal aparece na caixa de seleção. Caso contrário, a caixa de seleção permanece vazia. As caixas de seleção podem ser utilizadas individualmente e em grupos. As caixas de seleção que fazem parte do mesmo grupo têm o mesmo nome.
- Um botão de opção é semelhante em função e uso a uma caixa de seleção, exceto pelo fato de que apenas um botão de opção em um grupo pode ser selecionado por vez. Todos os botões de opção em um grupo têm o mesmo valor de atributo `name` e têm atributos `value` diferentes.
- A entrada `select` fornece uma lista drop-down de itens. O atributo `name` identifica a lista drop-down. O elemento `option` adiciona itens à lista drop-down. O atributo `selected`, como o atributo `checked` para botões de opção e caixas de seleção, especifica qual item da lista é exibido inicialmente.

## Terminologia

<code>&lt;!--...--&gt;</code> (comentário XHTML)	âncora de correio eletrônico	colchetes angulares ( <code>&lt; &gt;</code> )
<code>&amp;amp;</code> (caractere especial <code>&amp;</code> )	área de texto	<code>colgroup</code> , elemento
<code>&amp;copy;</code> (caractere especial ©)	atributo	<code>cols</code> , atributo
<code>.htm</code> (extensão do nome do arquivo XHTML)	<code>body</code> , elemento	<code>colspan</code> , atributo
<code>.html</code> (extensão do nome do arquivo XHTML)	<code>border</code> , atributo	comentário XHTML
"radio" (valor de atributo)	<code>br</code> , elemento (quebra de linha)	comentários XHTML
<code>&lt;caption&gt;</code> , tag	cabeçalho	declaração XML
<code>&lt;hr /&gt;</code> , tag (linha horizontal)	caixa de seleção	disco
<code>&lt;html&gt;</code> , tag	caixa de senha	documento vinculado
<code>&lt;strong&gt;</code> , tag	caractere especial	editor de texto
<code>&lt;thead&gt;...&lt;/thead&gt;</code>	célula de cabeçalho	elemento
<code>action</code> , atributo	<code>checked</code> , atributo	elemento a ( <code>&lt;a&gt;...&lt;/a&gt;</code> )
<code>alt</code> , atributo	código hexadecimal	elemento de lista não ordenada ( <code>ul</code> )
âncora	<code>col</code> , elemento	elementos nível de título ( <code>h1</code> a <code>h6</code> )

form, elemento	ol, elemento (lista ordenada)	tag vazio
formulário	p, elemento (parágrafo)	tag XHTML
formulário XHTML	página Web	tbody, elemento
head, elemento	rows, atributo (textarea)	td, elemento
height, atributo	rowspan, atributo (tr)	textarea, elemento
hidden input, elemento	selected, atributo	tfoot, elemento (rodapé de tabela)
href, atributo	servidor Web	title, elemento
hyperlink	sintaxe	tr, elemento (linha de tabela)
hyperlink de imagem	size, atributo (input)	type, atributo
img, elemento	sobrescrito	URL mail to:
input, elemento	solicitação de navegador	valign, atributo (th)
linguagem de marcação	src, atributo (img)	value, atributo
lista aninhada	sub, elemento	width, atributo
marcação XHTML	subscrito	World Wide Web (WWW)
maxlength, atributo	table, elemento	XHTML (Extensible Hypertext Markup Language)
method, atributo	tag	xmlns, atributo
name, atributo	tag    (item de lista)	
nível de aninhamento	tag aninhado	

## K

## Caracteres especiais de XHTML

A tabela da Figura K.1 mostra muitos caracteres especiais XHTML comumente utilizados — chamados de *referências de entidade de caractere* pelo World Wide Web Consortium. Para uma lista completa de referências de entidade de caractere, visite o site

[www.w3.org/TR/REC-html40/sgml/entities.html](http://www.w3.org/TR/REC-html40/sgml/entities.html)

Caractere	Codificação XHTML	Caractere	Codificação XHTML
Espaço não separável	&#160;	ê	&#234;
§	&#167;	ì	&#236;
©	&#169;	í	&#237;
®	&#174;	î	&#238;
π	&#188;	ñ	&#241;
∫	&#189;	ò	&#242;
Ω	&#190;	ó	&#243;
à	&#224;	ô	&#244;
á	&#225;	õ	&#245;
â	&#226;	÷	&#247;
ã	&#227;	ù	&#249;
ä	&#229;	ú	&#250;
ç	&#231;	û	&#251;
è	&#232;	•	&#8226;
é	&#233;	™	&#8482;

**Figura K.1** Caracteres especiais XHTML.



*E, então, eu devo pegar a mosca.*

William Shakespeare

*Fomos criados para cometer equívocos, programados para erro.*

Lewis Thomas

*O que antecipamos raramente ocorre; o que menos esperamos geralmente acontece.*

Benjamin Disraeli

*Ele pode correr, mas não pode se esconder.*

Joe Louis

*Uma coisa é mostrar a um homem que ele está errado, outra é colocá-lo de posse da verdade.*

John Locke

L

## Utilizando o depurador do Visual Studio .NET

### OBJETIVOS

Neste apêndice, você aprenderá:

- A configurar pontos de interrupção para depurar programas.
- A executar um programa pelo depurador.
- A configurar, desativar e remover um ponto de interrupção.
- Como utilizar o comando **Continue** para continuar a execução.
- Como utilizar a janela **Locals** para visualizar e modificar os valores de variáveis.
- Como utilizar a janela **Watch** para avaliar expressões.
- Como utilizar os comandos **Step Into**, **Step Out** e **Step Over** para controlar execução.
- Como utilizar a janela **Autos** para visualizar variáveis que são utilizadas em instruções envolventes.

- L.1** Introdução
- L.2** Pontos de interrupção e o comando Continue
- L.3** As janelas Locals e Watch
- L.4** Controlando a execução utilizando os comandos Step Into, Step Over, Step Out e Continue
- L.5** A janela Autos
- L.6** Síntese

Resumo | Terminologia | Exercícios de revisão | Respostas dos exercícios de revisão

## L.1 Introdução

No Capítulo 2, você aprendeu que há dois tipos de erro — erros de compilação e erros de lógica — e aprendeu a eliminar erros de sintaxe do seu código. Os erros de lógica (também chamado de **bugs**) não impedem que um programa compile com sucesso, mas fazem com que o programa produza resultados errôneos ao executar. A maioria dos fornecedores de compilador C++ oferece um software chamado **depurador**, que permite monitorar a execução de seus programas para localizar e remover erros de lógica. O depurador será uma das suas ferramentas mais importantes de desenvolvimento de programa. Este apêndice demonstra recursos-chave do depurador do Visual Studio .NET. O Apêndice M discute os recursos e capacidades do depurador do GNU C++. Fornecemos várias publicações *Dive Into™ Series* gratuitas para ajudar alunos e instrutores a familiarizar-se com os depuradores fornecidos com as várias ferramentas de desenvolvimento. Os links das publicações estão disponíveis no CD que acompanha o texto e seu download pode ser feito a partir de [www.deitel.com/books/downloads](http://www.deitel.com/books/downloads).

## L.2 Pontos de interrupção e o comando Continue

Iniciamos nosso estudo do depurador investigando os **pontos de interrupção**, marcadores que podem ser configurados em qualquer linha executável do código. Quando a execução do programa alcança um ponto de interrupção, a execução pausa, permitindo que você examine os valores das variáveis para ajudar a determinar se há erros de lógica. Por exemplo, você pode examinar o valor de uma variável que armazena o resultado de um cálculo a fim de determinar se o cálculo foi realizado corretamente. Observe que tentar configurar um ponto de interrupção em uma linha do código que não é executável (como um comentário) na realidade irá configurar o ponto de interrupção na próxima linha de código executável nessa função.

Para ilustrar os recursos do depurador, utilizamos a listagem de programa na Figura L.3, que cria e manipula um objeto da classe Account (figuras L.1–L.2). A execução inicia em main (linhas 12–30 da Figura L.3). A linha 14 cria um objeto Account com um saldo inicial de \$50,00. O construtor de Account (linhas 10–22 da Figura L.2) aceita um argumento, que especifica o balance inicial de Account. A linha 17 da Figura L.3 gera saída do balanço inicial da conta utilizando a função-membro Account::getBalance. A linha 19 declara uma variável local withdrawalAmount, que armazena a quantia de retirada lida a partir do usuário. A linha 21 solicita ao usuário a quantia de retirada e a linha 22 realiza a entrada da quantia em withdrawalAmount. A linha 25 subtrai a quantia retirada do balance de Account utilizando sua função-membro debit. Por fim, a linha 28 exibe o novo balance.

Nos passos indicados a partir da página L-4, você utilizará pontos de interrupção e vários comandos de depurador para examinar o valor da variável withdrawalAmount declarado na Figura L.3.

```

1 // Figura L.1: Account.h
2 // Definição da classe Account.
3
4 class Account
5 {
6 public:
7 Account(int); // o construtor inicializa balance
8 void credit(int); // adiciona uma quantia ao saldo da conta
9 void debit(int); // subtrai uma quantia do saldo da conta
10 int getBalance(); // retorna o saldo da conta
11 private:
12 int balance; // membro de dados que armazena o saldo
13 }; // fim da classe Account

```

**Figura L.1** Arquivo de cabeçalho da classe Account.

```

1 // Figura L.2: Account.cpp
2 // Definições de função-membro para a classe Account.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Account.h" // inclui a definição de classe Account
8
9 // construtor Account inicializa membro de dados balance
10 Account::Account(int initialBalance)
11 {
12 balance = 0; // assume que balance inicia em 0
13
14 // se initialBalance for maior que 0, configura esse valor como o
15 // saldo [balance] da Account; caso contrário, o balance retorna 0
16 if (initialBalance > 0)
17 balance = initialBalance;
18
19 // se initialBalance for negativo, imprime mensagem de erro
20 if (initialBalance < 0)
21 cout << "Error: Initial balance cannot be negative.\n" << endl;
22 } // fim do construtor Account
23
24 // credita (adiciona) uma quantia ao saldo da conta
25 void Account::credit(int amount)
26 {
27 balance = balance + amount; // adiciona amount ao balance
28 } // fim da função credit
29
30 // debita (subtrai) uma quantia do saldo da conta
31 void Account::debit(int amount)
32 {
33 if (amount <= balance) // a quantia de débito não excede o saldo
34 balance = balance - amount;
35
36 else // a quantia de débito excede o saldo
37 cout << "Debit amount exceeded account balance.\n" << endl;
38 } // fim da função debit
39
40 // retorna o saldo da conta
41 int Account::getBalance()
42 {
43 return balance; // fornece o valor do saldo à função chamadora
44 } // fim da função getBalance

```

**Figura L.2** Definição da classe Account.

```

1 // Figura L.3: figL_03.cpp
2 // Cria e manipula um objeto Account.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6 using std::endl;
7

```

**Figura L.3** Classe de teste para depuração.

(continua)

```

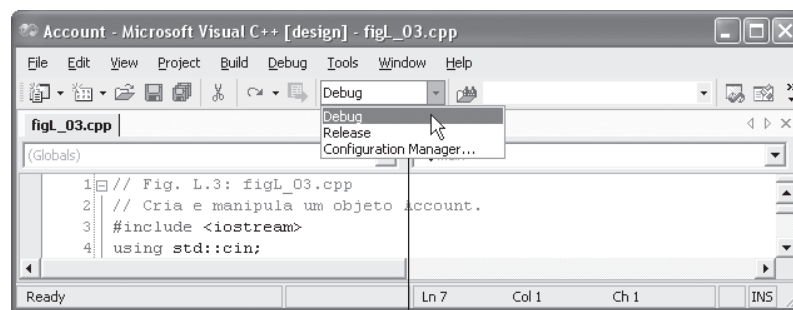
8 // inclui a definição de classe Account a partir de Account.h
9 #include "Account.h"
10
11 // a função main inicia a execução do programa
12 int main()
13 {
14 Account account1(50); // cria o objeto Account
15
16 // exibe saldo inicial de cada objeto
17 cout << "account1 balance: $" << account1.getBalance() << endl;
18
19 int withdrawalAmount; // armazena a quantia de depósito lida a partir do usuário
20
21 cout << "\nEnter withdrawal amount for account1: "; // prompt
22 cin >> withdrawalAmount; // obtém a entrada do usuário
23 cout << "\nattempting to subtract " << withdrawalAmount
24 << " from account1 balance\n\n";
25 account1.debit(withdrawalAmount); // tenta subtrair de account1
26
27 // exibe os saldos
28 cout << "account1 balance: $" << account1.getBalance() << endl;
29 return 0; // indica terminação bem-sucedida
30 } // fim do main

```

Figura L.3 Classe de teste para depuração.

(continuação)

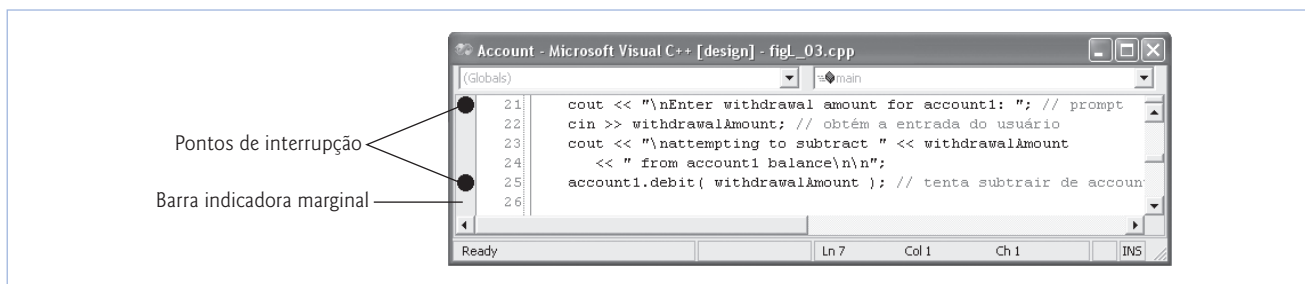
1. **Ativando o depurador.** O depurador está ativado por padrão. Se ele não estiver ativado, você tem de alterar as configurações da **caixa de combinação Solution Configurations** (Figura L.4) na barra de ferramentas. Para fazer isso, clique na seta para baixo da caixa de combinação a fim de acessar a caixa de combinação *Solution Configurations*, então selecione Debug. A barra de ferramentas exibirá Debug na caixa de combinação *Solution Configurations*.
2. **Inserindo pontos de interrupção no Visual Studio .NET.** Para inserir um ponto de interrupção no Visual Studio .NET, clique dentro da **barra indicadora marginal** (a margem acinzentada à esquerda da janela de código na Figura L.5) ao lado da linha de código em que você deseja quebrar ou clique com o botão direito do mouse nessa linha de código e selecione Insert Breakpoint. Você pode configurar quantos pontos de interrupção forem necessários. Configure pontos de interrupção nas linhas 21 e 25 de seu código. Um círculo sólido marrom aparece em sua tela (aqui você vê em preto), na barra indicadora marginal onde você clicou, indicando que um ponto de interrupção foi configurado (Figura L.5). Quando o programa executa, o depurador suspende a execução em qualquer linha que contenha um ponto de interrupção. Diz-se que o programa está no **modo de interrupção (break mode)** quando o depurador pausa a execução do programa. Os pontos de interrupção podem ser configurados antes de executar um programa, no modo de interrupção e enquanto um programa está executando.



A caixa de combinação Solution Configurations

Figura L.4 Ativando o depurador.



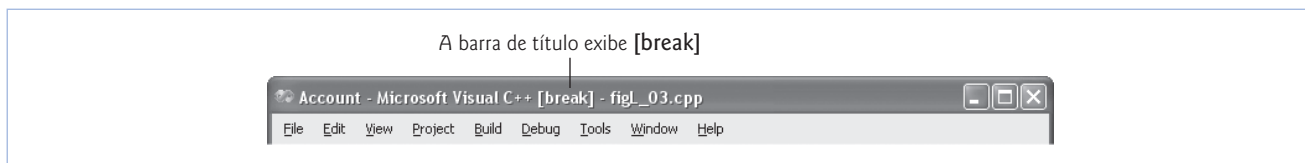


**Figura L.5** Configurando dois pontos de interrupção.

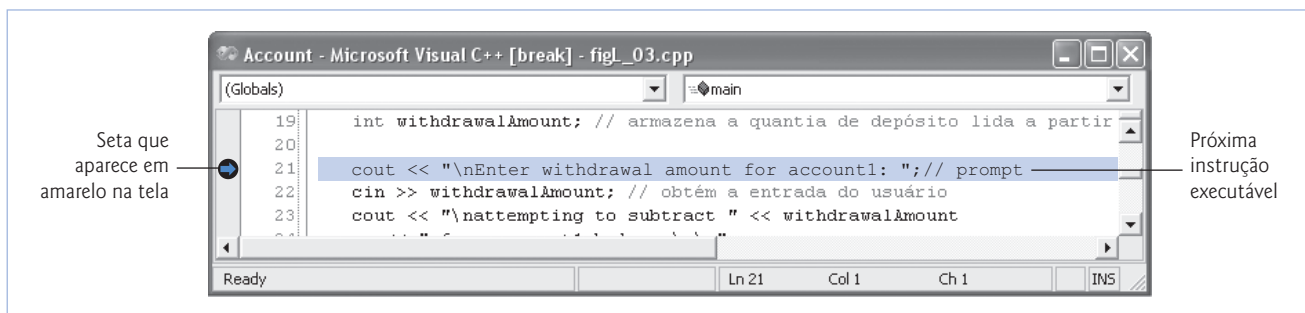
3. *Começando o processo de depuração.* Depois de configurar pontos de interrupção no editor de código, selecione Build > Build Solution para compilar o programa, então selecione Debug > Start para iniciar o processo de depuração. Durante a depuração de um programa C++, uma janela Command Prompt aparece (Figura L.6), permitindo interação com o programa (entrada e saída). O programa pausa quando a execução alcança o ponto de interrupção na linha 21. Nesse ponto, a barra de título do IDE exibirá [break] (Figura L.7), indicando que o IDE está no modo de interrupção.
4. *Examinando a execução do programa.* A execução do programa é suspensa no primeiro ponto de interrupção (linha 21) e o IDE torna-se a janela ativa (Figura L.8). A **seta amarela** (que você vê em azul na figura) à esquerda da linha 21 indica que essa linha contém a próxima instrução a executar. [Nota: Adicionamos o azul para destacar a linha do código. Seu código não conterá esse destaque.]
5. *Utilizando o comando Continue para retomar a execução.* Para retomar a execução, selecione Debug > Continue. O comando Continue executará quaisquer instruções entre a próxima instrução executável e o próximo ponto de interrupção ou o fim do main, o que vier primeiro. O programa continua a executar e pausa para entrada na linha 22. Insira 13 como a quantia de retirada. O programa executa até parar no próximo ponto de interrupção, linha 25. Observe que, quando você coloca o ponteiro



**Figura L.6** O programa **Inventory** em execução.



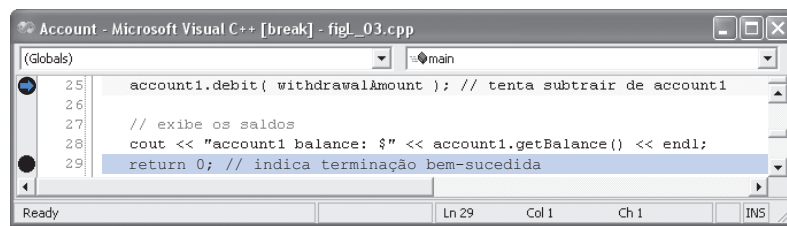
**Figura L.7** A barra de título do IDE exibe [break].



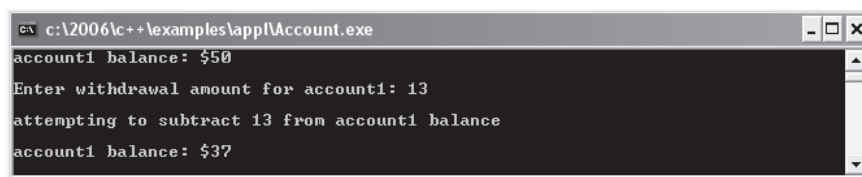
**Figura L.8** Execução do programa suspensa no primeiro ponto de interrupção.

do mouse sobre o nome da variável `withdrawalAmount`, o valor que ela armazena é exibido em uma **caixa Quick Info** (Figura L.9). Em certo sentido, você está espiando dentro do computador o valor de uma de suas variáveis. Como você verá, isso pode ajudá-lo a identificar erros de lógica em seus programas.

6. *Configurando um ponto de interrupção na instrução **return**.* Configure um ponto de interrupção na linha 29 no código-fonte clicando na barra indicadora marginal à esquerda da linha 29 (Figura L.9). Isso impedirá que o programa se feche imediatamente depois de exibir seu resultado. Quando não houver mais pontos de interrupção nos quais suspender execução, o programa executará até sua conclusão e a janela **Command Prompt** se fechará. Se você não configurar esse ponto de interrupção, não será capaz de ver a saída do programa antes de a janela de console se fechar.
7. *Continuando a execução do programa.* Utilize o comando **Debug > Continue** para executar a linha 25. O programa exibe o resultado de seu cálculo (Figura L.10).
8. *Desativando um ponto de interrupção.* Para **desativar um ponto de interrupção**, dê um clique com o botão direito do mouse em uma linha do código na qual um ponto de interrupção foi configurado (ou no próprio ponto de interrupção) e selecione **Disable Breakpoint**. O ponto de interrupção desativado é indicado na tela por um círculo marrom vazio (círculo preto vazio na Figura L.11). Desativar em vez de remover um ponto de interrupção permite reativar o ponto de interrupção (clicando dentro do círculo vazio) em um programa. Isso também pode ser feito clicando com o botão direito do mouse na linha marcada pelo círculo marrom (ou no próprio círculo) e selecionando **Enable Breakpoint**.
9. *Removendo um ponto de interrupção.* Para remover um ponto de interrupção de que você não precisa mais, dê um clique com o botão direito do mouse em uma linha do código na qual um ponto de interrupção foi configurado e selecione **Remove Breakpoint**. Você também pode remover um ponto de interrupção clicando no círculo marrom na barra indicadora marginal.
10. *Finalizando a execução do programa.* Selecione **Debug > Continue** para executar o programa até a conclusão.

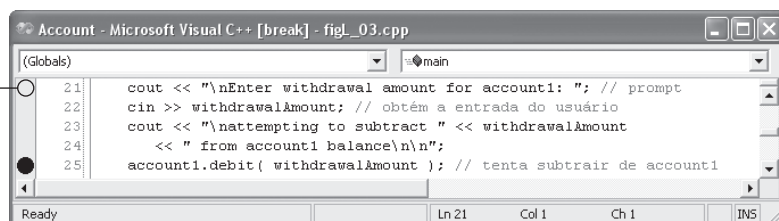


**Figura L.9** Configurando um ponto de interrupção na linha 29.



**Figura L.10** Program output.

Ponto de interrupção desativado



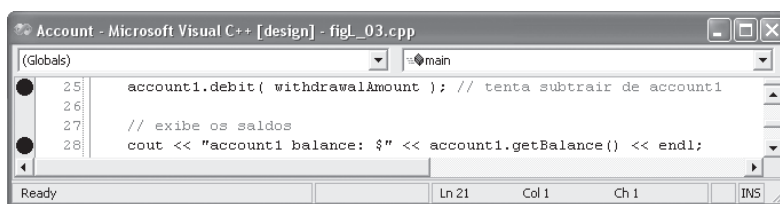
**Figura L.11** Ponto de interrupção desativado.

Nesta seção, você aprendeu a ativar o depurador e a configurar pontos de interrupção de modo a poder examinar os resultados do código enquanto um programa estava executando. Você também aprendeu a continuar a execução depois que um programa suspende a execução em um ponto de interrupção e a desativar e remover pontos de interrupção.

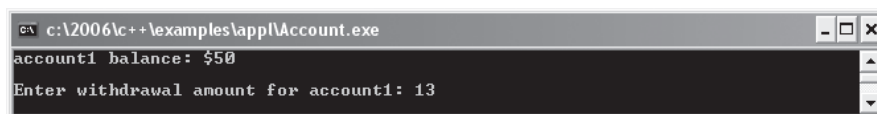
## L.3 As janelas Locals e Watch

Na seção precedente, você aprendeu que o recurso *Quick Info* permite examinar o valor de uma variável. Nesta seção, você aprenderá a utilizar a **janela Locals** para atribuir novos valores a variáveis enquanto seu programa está executando. Você também utilizará a **janela Watch** para examinar o valor de expressões mais complexas.

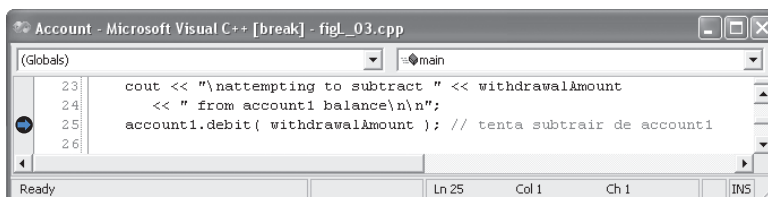
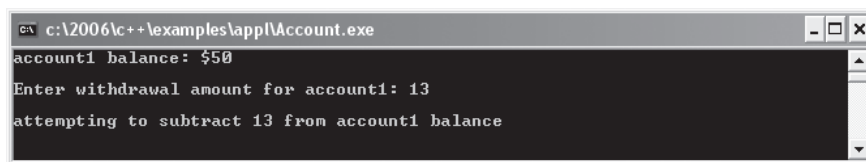
1. *Inserindo pontos de interrupção.* Configure um ponto de interrupção na linha 25 do código-fonte clicando na barra indicadora marginal à esquerda da linha 25 (Figura L.12). Configure outro ponto de interrupção na linha 28 do código clicando na barra indicadora marginal à esquerda da linha 28.
2. *Iniciando a depuração.* Selecione **Debug > Start**. Digite 13 no prompt **Enter withdrawal amount for account1:** (Figura L.13) e pressione **Enter** para seu programa ler o valor que você acaba de inserir. O programa executa até o ponto de interrupção na linha 25.
3. *Suspendendo a execução de programa.* Quando o programa alcança a linha 25, o Visual C++ .NET suspende a execução do programa e alterna o programa para o modo de interrupção (Figura L.14). Nesse ponto, a instrução na linha 22 (Figura L.3) aceitou a entrada de `withdrawalAmount` que você inseriu (13), a instrução na linha 23–24 gerou uma saída informando que o programa tentará retirar dinheiro, e a instrução na linha 25 é a próxima instrução que será executada.



**Figura L.12** Configurando pontos de interrupção nas linhas 25 e 28.

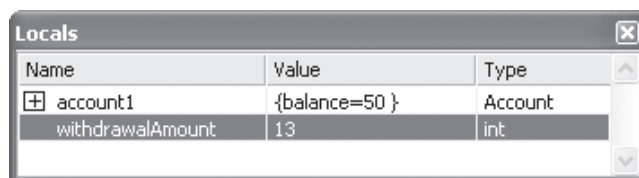


**Figura L.13** A quantia de retirada que entra antes de o ponto de interrupção ser alcançado.

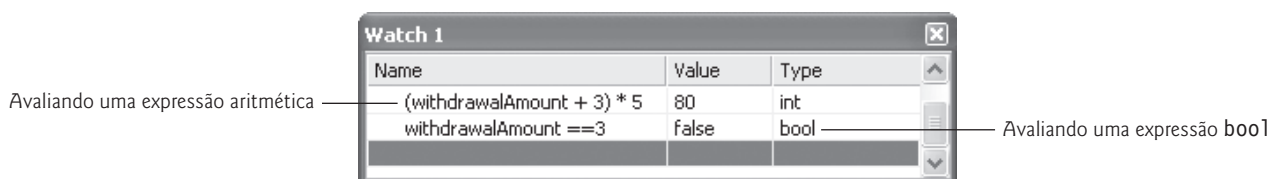


**Figura L.14** A execução do programa suspende quando o depurador alcança o ponto de interrupção na linha 25.

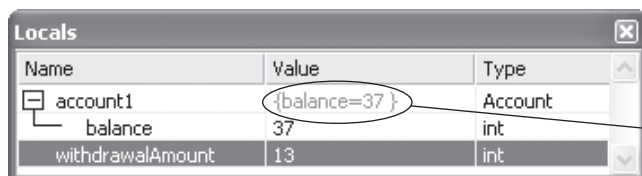
4. *Examinando os dados.* Uma vez que o programa entrou no modo de interrupção, você pode explorar os valores de suas variáveis locais utilizando a janela Locals do depurador. Para visualizar a janela Locals, selecione Debug > Windows > Locals. Os valores para `account1` e `withdrawalAmount` (13) são exibidos (Figura L.15).
5. *Avaliando expressões aritméticas e booleanas.* O Visual Studio .NET permite avaliar expressões aritméticas e booleanas utilizando a janela Watch. Há quatro janelas Watch diferentes, mas utilizaremos somente a primeira janela. Selecione Debug > Windows > Watch > Watch 1. Na primeira linha da coluna Name (que deve estar em branco inicialmente), digite `(withdrawalAmount + 3) * 5`, então pressione *Enter*. Observe que a janela Watch pode avaliar expressões aritméticas. Nesse caso, ela exibe o valor 80 (Figura L.16). Na próxima linha da coluna Name na janela Watch, digite `withdrawalAmount == 3` e então pressione *Enter*. Essa expressão determina se o valor contido em `withdrawalAmount` é 3. Expressões contendo o símbolo `==` são tratadas como expressões booleanas. O valor retornado é `false` (Figura L.16), porque `withdrawalAmount` atualmente não contém o valor 3.
6. *Retomando a execução.* Selecione Debug > Continue para retomar a execução. A linha 25 executa, debitando da conta a quantia de retirada, e o programa mais uma vez é suspenso na linha 28. Selecione Debug > Windows > Locals. O valor atualizado de `account1` agora é exibido na tela em vermelho para indicar que ele foi modificado desde o último ponto de interrupção (destacado por um círculo na Figura L.17). O valor em `withdrawalAmount` não está em vermelho porque não foi atualizado desde o último ponto de interrupção. Clique na caixa com um sinal de adição à esquerda de `account1` na coluna Name da janela Locals. Isso permite visualizar cada um dos valores do membro de dados `account1` individualmente.
7. *Modificando valores.* Com base no valor inserido pelo usuário (13), a saída do saldo da conta pelo programa deve ser \$37. Entretanto, você pode utilizar o depurador para alterar os valores de variáveis no meio da execução do programa. Isso pode ser valioso para experimentar diferentes valores e localizar erros de lógica nos programas. Você pode utilizar a janela Locals para alterar o valor de uma variável. Na janela Locals, clique no campo Value na linha `balance` para selecionar o valor 37. Digite 33 e então pressione *Enter*. O depurador altera o valor de `balance` e exibe seu novo valor em vermelho (veja o destaque com o círculo na Figura L.18).



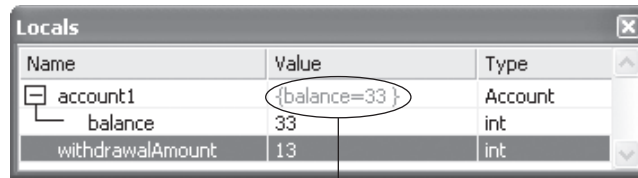
**Figura L.15** Examinando a variável `withdrawalAmount`.



**Figura L.16** Examinando os valores das expressões.



**Figura L.17** Exibindo o valor de variáveis locais.



Name	Value	Type
account1	{balance=33}	Account
balance	33	int
withdrawalAmount	13	int

Valor modificado no depurador

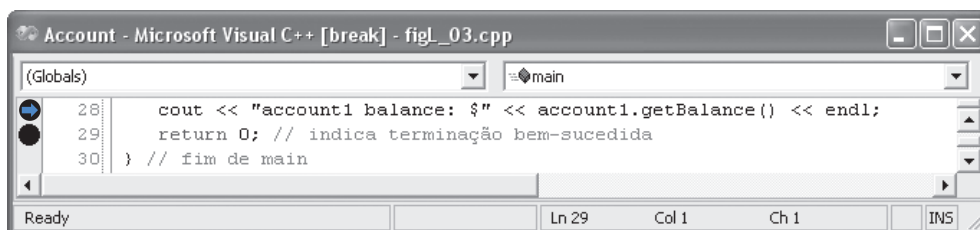
**Figura L.18** Modificando o valor de uma variável.

8. *Configurando um ponto de interrupção na instrução **return**.* Configure um ponto de interrupção na linha 29 no código-fonte clicando na barra indicadora marginal à esquerda da linha 29 (Figura L.19). Isso impedirá que o programa se feche imediatamente depois de exibir seu resultado. Se você não configurou esse ponto de interrupção, não será capaz de ver a saída do programa antes de a janela de console se fechar.
9. *Visualizando o resultado do programa.* Selecione Debug > Continue para continuar a execução do programa. A função main executa até a instrução return na linha 29 e exibe o resultado. Observe que o resultado é \$33 (Figura L.20). Isso mostra que o passo anterior alterou o valor de balance em relação ao valor calculado (37) para 33.
10. *Parando a sessão de depuração.* Selecione Debug > Stop Debugging. Isso fechará a janela Command Prompt. Remova todos os pontos de interrupção restantes.

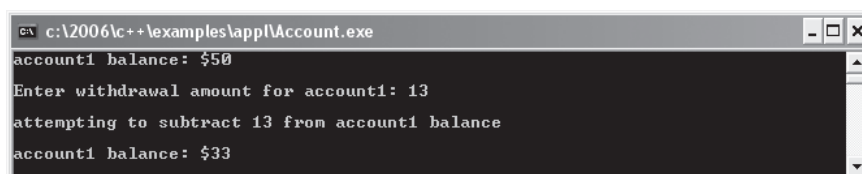
Nesta seção, você aprendeu a utilizar as janelas Watch e Locals do depurador para avaliar expressões aritméticas e booleanas. Você também aprendeu a modificar o valor de uma variável durante a execução do seu programa.

## L.4 Controlando a execução utilizando os comandos Step Into, Step Over, Step Out e Continue

Às vezes você precisará executar um programa linha por linha para localizar e corrigir erros de lógica. Inspeccionar passo a passo uma parte do seu programa dessa maneira pode ajudá-lo a verificar se o código de uma função executa corretamente. Nesta seção, você aprenderá a utilizar o depurador para essa tarefa. Os comandos que você aprende nesta seção lhe permitem executar uma função linha por linha, executar todas as instruções de uma função de uma vez ou executar somente as instruções restantes de uma função (se você já executou algumas instruções dentro da função).



**Figura L.19** Configurando um ponto de interrupção na linha 29.



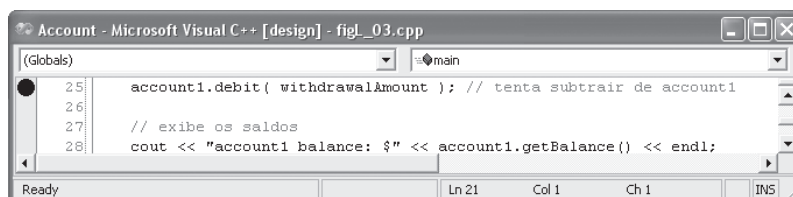
```

c:\2006\c++\examples\appt\Account.exe
account1 balance: $50
Enter withdrawal amount for account1: 13
attempting to subtract 13 from account1 balance
account1 balance: $33

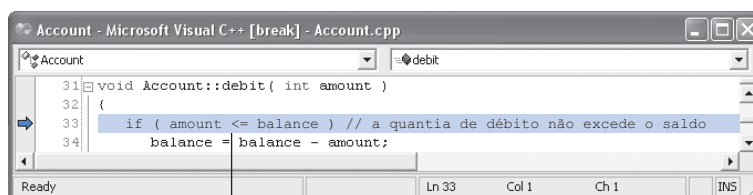
```

**Figura L.20** Saída exibida depois de modificar a variável account1.

1. *Configurando um ponto de interrupção.* Configure um ponto de interrupção na linha 25 clicando na barra indicadora marginal (Figura L.21).
2. *Iniciando o depurador.* Selecione Debug > Start. Insira o valor 13 no prompt Enter withdrawal amount for account1:. A execução parará quando o programa alcançar o ponto de interrupção na linha 25.
3. *Utilizando o comando Step Into.* O **comando Step Into** executa a próxima instrução no programa (a linha destacada em azul na Figura L.24) e imediatamente pára. Se a instrução a ser executada como resultado do comando Step Into for uma chamada de função, o controle é transferido à função chamada. O comando Step Into permite entrar em uma função e confirmar sua execução executando individualmente cada instrução dentro da função. Selecione Debug > Step Into para entrar na função `debit` (Figura L.22). Se o depurador não estiver na linha 33, selecione Debug > Step Into novamente para alcançar essa linha.
4. *Utilizando o comando Step Over.* Selecione Debug > Step Over para executar a instrução atual (linha 33 na Figura L.22) e transferir o controle para a linha 34 (Figura L.23). O **comando Step Over** comporta-se como o comando Step Into quando a próxima instrução a executar não contém uma chamada de função. Você verá em que o comando Step Over difere do comando Step Into no Passo 10.
5. *Utilizando o comando Step Out.* Selecione Debug > Step Out para executar as instruções restantes na função e retornar o controle para a próxima instrução executável (linha 28 na Figura L.3), que contém a chamada de função. Frequentemente, em funções prolongadas, você vai querer ver algumas linhas de código-chave e então continuar a depurar o código do chamador. O **comando Step Out** é útil para tais situações, em que você não quer continuar a inspecionar a função inteira linha por linha.
6. *Configurando um ponto de interrupção.* Configure um ponto de interrupção (Figura L.25) na instrução `return` de `main` na linha 29 da Figura L.3. Você utilizará esse ponto de interrupção no próximo passo.

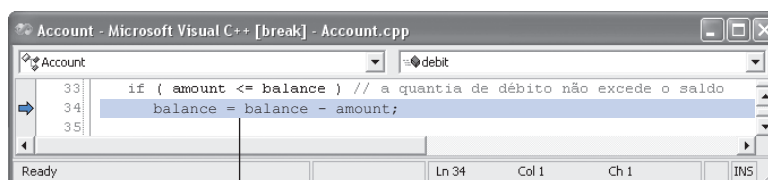


**Figura L.21** Configurando um ponto de interrupção no programa



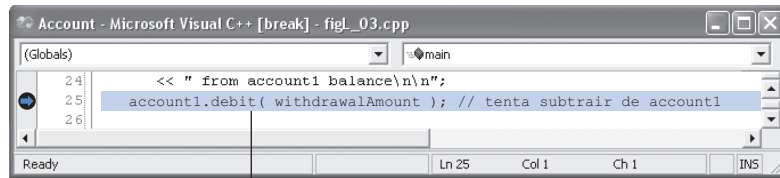
Próxima instrução a executar

**Figura L.22** Inspeccionando passo a passo (*stepping into*) a função `debit`.



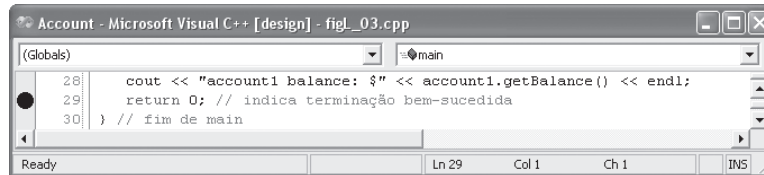
O controle é transferido para a próxima instrução

**Figura L.23** Inspeção com transferência do controle (*stepping over*) para uma instrução na função `debit`.



Próxima instrução executar é uma chamada de função

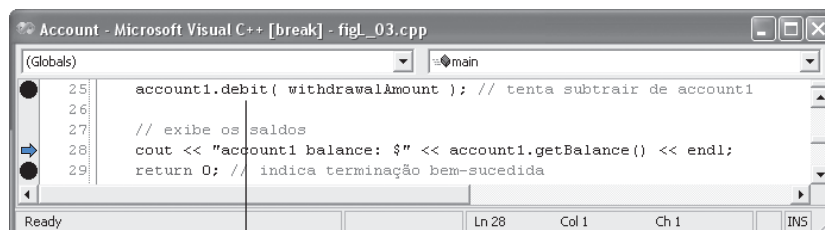
**Figura L.24** Utilizando o comando **Step Into** para executar uma instrução.



**Figura L.25** Configurando um segundo ponto de interrupção no programa.

7. *Utilizando o comando Continue.* Selecione **Debug > Continue** para executar até o próximo ponto de interrupção ser alcançado na linha 29. Este recurso economiza tempo quando você não quer inspecionar linha por linha um código extenso para alcançar o próximo ponto de interrupção.
8. *Parando o depurador.* Selecione **Debug > Stop Debugging** para encerrar a sessão de depuração. Isso fechará a janela **Command Prompt**.
9. *Iniciando o depurador.* Antes de podermos demonstrar o próximo recurso de depurador, você deve iniciar o depurador novamente. Inicie-o, como fez no Passo 2, e insira como entrada o mesmo valor (13). O depurador pausa a execução na linha 25.
10. *Utilizando o comando Step Over.* Selecione **Debug > Step Over** (Figura L.26). Lembre-se de que esse comando se comporta como o comando **Step Into** quando a próxima instrução a executar não contém uma chamada de função. Se a próxima instrução a executar contiver uma chamada de função, a função chamada executa em sua totalidade (sem pausar a execução em nenhuma instrução dentro da função) e a seta amarela (que aqui você vê em azul) avança para a próxima linha executável (depois da chamada de função) na função atual. Nesse caso, o depurador executa a linha 25, localizada em `main` (Figura L.3). A linha 25 chama a função `debit`. O depurador então pausa a execução na linha 28, a próxima linha executável na função atual, `main`.
11. *Parando o depurador.* Selecione **Debug > Stop Debugging**. Isso fechará a janela **Command Prompt**. Remova todos os pontos de interrupção restantes.

Nesta seção, você aprendeu a utilizar o comando **Step Into** do depurador para depurar chamadas de função durante a execução do seu programa. Você viu como o comando **Step Over** pode ser utilizado para a inspeção pular (*step over*) uma chamada de função. Você viu como o comando **Step Out** pode ser utilizado para continuar a execução até o fim da função atual. Também aprendeu que o comando **Continue** continua a execução até que outro ponto de interrupção seja encontrado ou o programa encerre.



A função `debit` é executada sem que a inspeção entre nela quando o comando **Step Over** é selecionado

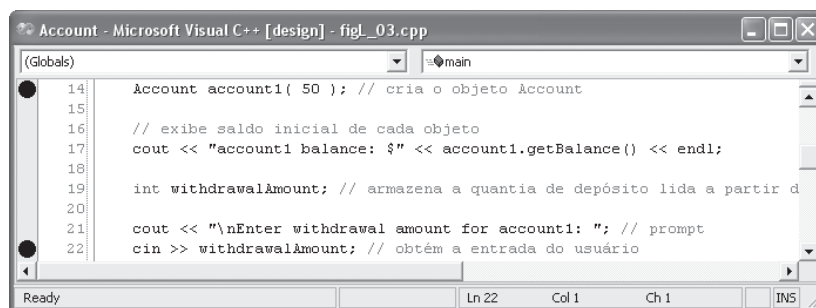
**Figura L.26** Utilizando o comando **Step Over** do depurador.



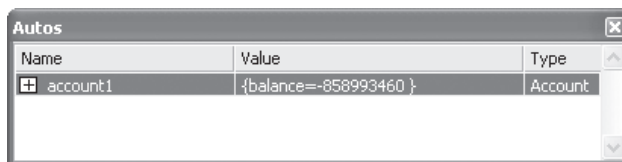
## L.5 A janela Autos

Nesta seção, apresentamos a **janela Autos**, que exibe as variáveis utilizadas na instrução anterior executada e o próximo comando a executar. A janela Autos permite focalizar variáveis que acabaram de ser utilizadas e aquelas que serão utilizadas e modificadas na próxima instrução.

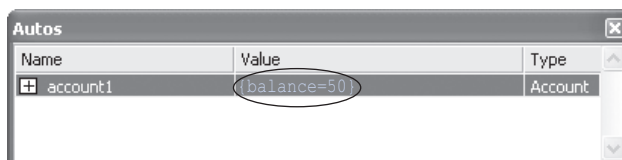
1. *Configurando pontos de interrupção.* Configure pontos de interrupção nas linhas 14 e 22 clicando na barra indicadora marginal (Figura L.27).
2. *Utilizando a janela Autos.* Inicie o depurador selecionando Debug > Start. Quando a execução pára no ponto de interrupção na linha 14, abra a janela Autos (Figura L.28) selecionando Debug > Windows > Autos. A janela Autos permite visualizar o conteúdo das variáveis utilizadas na última instrução que foi executada. Isso permite verificar se a instrução anterior executou corretamente. A janela Autos também lista os valores na próxima instrução a ser executada. Observe que a janela Autos lista a variável `account1`, seu valor e seu tipo. Visualizar os valores armazenados em um objeto permite-lhe verificar se seu programa está manipulando essas variáveis corretamente. Observe que `account1` contém um valor negativo grande. Esse valor, que pode ser diferente toda vez o programa executa, é o valor não inicializado de `account1`. Esse valor imprevisível (e freqüentemente indesejável) demonstra por que é importante inicializar todas as variáveis C++ antes de utilizá-las.
3. *Utilizando o comando Step Over.* Selecione Debug > Step Over para executar a linha 14. A janela Autos (Figura L.29) atualiza o valor de `account1` depois que essa variável é inicializada. O valor de `account1` é exibido em vermelho (aqui, destacado por um círculo) para indicar que foi alterado.
4. *Continuando a execução.* Selecione Debug > Continue. A execução do programa irá parar no segundo ponto de interrupção, configurado na linha 22. A janela Autos (Figura L.30) exibe a variável local não inicializada `withdrawalAmount`, que tem um valor negativo grande.



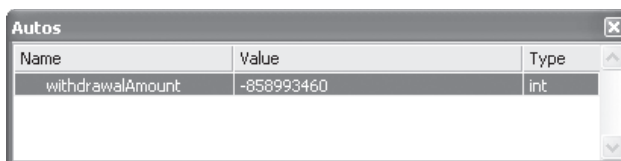
**Figura L.27** Configurando pontos de interrupção no programa.



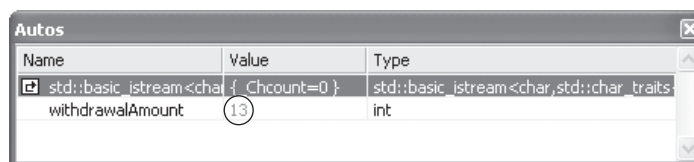
**Figura L.28** A janela **Autos** exibe o estado do objeto `account1`.



**Figura L.29** A janela **Autos** exibe o estado do objeto `account1` depois da inicialização.



**Figura L.30** A janela **Autos** exibe a variável local `withdrawalAmount`.



**Figura L.31** A janela **Autos** exibe a variável local `withdrawalAmount` atualizada.

5. *Inserindo dados.* Selecione **Debug > Step Over** para executar a linha 22. No prompt de entrada do programa, insira um valor para a quantia de retirada. A janela **Autos** (Figura L.29) atualizará o valor da variável local `withdrawalAmount` com o valor que você inseriu. [Nota: A primeira linha da janela **Autos** contém o objeto `istream (cin)` que você utilizou para inserir dados.]
6. *Parando o depurador.* Selecione **Debug > Stop Debugging** para encerrar a sessão de depuração. Remova todos os pontos de interrupção restantes.

Nesta seção, você aprendeu sobre a janela **Autos**, que lhe permite visualizar as variáveis utilizadas no comando mais recente.

## L.6 Síntese

Neste apêndice, você aprendeu a inserir, desativar e remover pontos de interrupção no depurador do Visual Studio .NET. Os pontos de interrupção permitem pausar a execução de um programa de modo que você possa examinar valores de variáveis. Essa capacidade o ajudará a localizar e corrigir erros de lógica nos seus programas. Você viu como utilizar as janelas **Locals** e **Watch** para examinar o valor de uma expressão e alterar o valor de uma variável. Você também aprendeu os comandos **Step Into**, **Step Over**, **Step Out** e **Continue** do depurador que podem ser utilizados para determinar se uma função está executando corretamente. Por fim, aprendeu a utilizar a janela **Autos** para examinar variáveis utilizadas especificamente nos comandos anteriores e nos próximos comandos.

### Resumo

- A maioria dos fornecedores de compilador C++ oferece um software chamado depurador, que lhe permite monitorar a execução de seus programas para localizar e remover erros de lógica.
- Os pontos de interrupção são marcadores que podem ser configurados em qualquer linha de código executável. Quando a execução do programa alcança um ponto de interrupção, a execução pausa.
- O depurador está ativado por padrão. Se ele não estiver ativado, você tem de alterar as configurações da caixa de combinação *Solution Configurations*.
- Para inserir um ponto de interrupção, clique dentro da barra indicadora marginal ao lado da linha de código ou dê um clique com o botão direito do mouse nessa linha de código e selecione **Insert Breakpoint**. Um círculo marrom sólido (que aqui no livro você vê em preto) aparece onde você clicou, indicando que um ponto de interrupção foi configurado.
- Quando o programa executa, ele suspende a execução em qualquer linha que contenha um ponto de interrupção. Dizemos então que o programa está no modo de interrupção e a barra de título do IDE exibirá **[break]**.
- Uma seta amarela (que aqui no livro você vê em azul) indica que essa linha contém a próxima instrução a executar.
- Quando você coloca o ponteiro do mouse sobre o nome da variável, o valor que a variável armazena é exibido em uma caixa *Quick Info*.
- Para desativar um ponto de interrupção, dê um clique com o botão direito do mouse em uma linha do código em que um ponto de interrupção foi configurado e selecione **Disable Breakpoint**. O ponto de interrupção desativado é indicado na tela por um círculo marrom vazio (no livro, em preto vazio).

- Para remover um ponto de interrupção de que você não precisa mais, dê um clique com o botão direito do mouse em uma linha do código em que um ponto de interrupção foi configurado e selecione **Remove Breakpoint**. Você também pode remover um ponto de interrupção clicando no círculo marrom que aparece na tela, na barra indicadora marginal.
- Uma vez que o programa entrou no modo de interrupção, você pode explorar os valores de suas variáveis utilizando a janela **Locals** do depurador. Para visualizar a janela **Locals**, selecione **Debug > Windows > Locals**.
- Você pode avaliar expressões aritméticas e booleanas utilizando as janelas **Watch**. A primeira janela **Watch** é exibida selecionando **Debug > Windows > Watch > Watch 1**.
- Variáveis atualizadas são exibidas na tela em vermelho para indicar que foram modificados desde o último ponto de interrupção.
- Clicar na caixa ao lado de um objeto na coluna **Name** da janela **Locals** permite visualizar individualmente cada um dos valores do membro de dados do objeto.
- Você pode clicar no campo **Value** de uma variável para alterar seu valor na janela **Locals**.
- O comando **Step Into** executa a próxima instrução (a linha azul destacada) no programa. Se a próxima instrução é executar uma chamada de função e você seleciona **Step Into**, o controle é transferido para a função chamada.
- O comando **Step Over** comporta-se como o comando **Step Into** quando a próxima instrução a executar não contém uma chamada de função. Se a próxima instrução a executar contiver uma chamada de função, a função chamada executa em sua totalidade e a seta amarela avança para a próxima linha executável na função atual.
- Selecione **Debug > Step Out** para executar as instruções restantes na função e retornar o controle para a chamada de função.
- O comando **Continue** executará quaisquer instruções entre a próxima instrução executável e o próximo ponto de interrupção ou o fim de `main`, o que vier primeiro.
- A janela **Autos** permite visualizar o conteúdo das variáveis utilizadas na última instrução que foi executada. A janela **Autos** também lista os valores na próxima instrução a ser executada.

## Terminologia

<b>Autos</b> , janela	<b>Locals</b> , janela	<b>Step Into</b> , comando
barra indicadora marginal	modo de interrupção	<b>Step Out</b> , comando
bug	ponto de interrupção	<b>Step Over</b> , comando
<b>Continue</b> , comando	<i>Quick Info</i> , caixa	<b>Watch</b> , janela
depurador	seta amarela no modo de interrupção	
desativando um ponto de interrupção	<i>Solution Configurations</i> , caixa de combinação	

## Exercícios de revisão

- L.1** Preencha as lacunas em cada uma das seguintes sentenças:
- Quando o depurador suspende a execução do programa em um ponto de interrupção, dizemos que o programa está no modo de \_\_\_\_\_.
  - O recurso \_\_\_\_\_ do Visual Studio .NET permite ‘olhar dentro do computador’ e ver o valor de uma variável.
  - Você pode examinar o valor de uma expressão utilizando a janela \_\_\_\_\_ do depurador.
  - O comando \_\_\_\_\_ comporta-se como o comando **Step Into** quando a próxima instrução a executar não contém uma chamada de função.
- L.2** Determine se cada uma das seguintes sentenças é *verdadeira* ou *falsa*. Se *falsa*, explique por quê.
- Quando a execução do programa é suspensa em um ponto de interrupção, a próxima instrução a ser executada é a instrução depois do ponto de interrupção.
  - Quando o valor de uma variável é alterado, ela se torna amarela nas janelas **Autos** e **Locals**.
  - Durante uma depuração, o comando **Step Out** executa as instruções restantes na função atual e retorna o controle do programa para o lugar onde a função foi chamada.

## Respostas dos exercícios de revisão

- L.1** a) interrupção. b) caixa *Quick Info*. c) **Watch**. d) **Step Over**.
- L.2** a) Falsa. Quando a execução do programa é suspensa em um ponto de interrupção, a próxima instrução a ser executada é a instrução no ponto de interrupção. b) Falsa. Uma variável torna-se vermelha quando seu valor é alterado. c) Verdadeira.

# M



*E, então, eu devo pegar a mosca.*

William Shakespeare

*Ele pode correr, mas não pode se esconder.*

Joe Louis

*Uma coisa é mostrar a um homem que ele está errado, outra é colocá-lo de posse da verdade.*

John Locke

*Fomos criados para cometer equívocos, programados para erro.*

Lewis Thomas

*O que antecipamos raramente ocorre; o que menos esperamos geralmente acontece.*

Benjamin Disraeli

## Utilizando o depurador do GNU C++

### OBJETIVOS

Neste apêndice, você aprenderá:

- Como utilizar o comando `run` para executar um programa no depurador.
- Como utilizar o comando `break` para configurar um ponto de interrupção.
- Como utilizar o comando `continue` para continuar a execução.
- Como utilizar o comando `print` para avaliar expressões.
- Como utilizar o comando `set` para alterar valores de variáveis durante a execução do programa.
- Como utilizar os comandos `step`, `finish` e `next` para controlar a execução.
- Como utilizar o comando `watch` para ver como um membro de dados é modificado durante a execução de um programa.
- Como utilizar o comando `delete` para remover um ponto de interrupção ou um ponto de observação.

- M.1** Introdução
- M.2** Pontos de interrupção e os comandos run, stop, continue e print
- M.3** Os comandos print e set
- M.4** Controlando a execução utilizando os comandos step, finish e next
- M.5** O comando watch
- M.6** Síntese

Resumo | Terminologia | Exercícios de revisão | Respostas dos exercícios de revisão

## M.1 Introdução

No Capítulo 2, você aprendeu que há dois tipos de erro — erros de compilação e erros de lógica — e aprendeu a eliminar erros de sintaxe do seu código. Os erros de lógica não impedem que um programa compile com sucesso, mas fazem com que o programa produza resultados errôneos quando executar. O GNU inclui software chamado **depurador** que permite monitorar a execução de seus programas de modo que você possa localizar e remover erros de lógica.

O depurador será uma das suas ferramentas mais importantes de desenvolvimento de programa. Muitos IDEs fornecem seus próprios depuradores semelhantes ao incluído no GNU ou fornecem uma interface gráfica com o usuário para o depurador do GNU. Este apêndice demonstra recursos-chave do depurador do GNU. O Apêndice L discute os recursos e as capacidades do depurador do Visual Studio .NET. Fornecemos várias publicações *Dive Into™ Series* gratuitas para ajudar alunos e instrutores a familiarizar-se com os depuradores fornecidos com várias ferramentas de desenvolvimento. Os links das publicações estão disponíveis no CD que acompanha o texto e seu download pode ser feito a partir de [www.deitel.com/books/downloads](http://www.deitel.com/books/downloads).

## M.2 Pontos de interrupção e os comandos run, stop, continue e print

Iniciamos nosso estudo do depurador investigando os **pontos de interrupção**, marcadores que podem ser configurados em qualquer linha executável do código. Quando a execução do programa alcança um ponto de interrupção, a execução pausa, permitindo que você examine os valores das variáveis para ajudar a determinar se há erros de lógica. Por exemplo, você pode examinar o valor de uma variável que armazena o resultado de um cálculo a fim de determinar se o cálculo foi realizado corretamente. Observe que tentar configurar um ponto de interrupção em uma linha de código que não é executável (como um comentário) na realidade irá configurar o ponto de interrupção na próxima linha de código executável nessa função.

Para ilustrar os recursos do depurador, utilizamos a listagem de programa na Figura M.3, que cria e manipula um objeto da classe Account (figuras M.1–M.2). A execução inicia em main (linhas 12–30 da Figura M.3). A linha 14 cria um objeto Account com um saldo inicial de \$50,00. O construtor de Account (linhas 10–22 da Figura M.2) aceita um argumento, que especifica o balance inicial de Account. A linha 17 da Figura M.3 gera saída do balanço inicial de conta utilizando a função-membro Account getBalance. A linha 19 declara uma variável local withdrawalAmount que armazena uma quantia de retirada lida a partir da entrada do usuário. A linha 21 pede para o usuário informar a quantia de retirada; a linha 22 realiza a entrada da quantia em withdrawalAmount. A linha 25 subtrai a quantia retirada do balance de Account utilizando sua função-membro debit. Por fim, a linha 28 exibe o novo balance.

```

1 // Figura M.1: Account.h
2 // Definição da classe Account.
3
4 class Account
5 {
6 public:
7 Account(int); // o construtor inicializa balance
8 void credit(int); // adiciona uma quantia ao saldo da conta
9 void debit(int); // subtrai uma quantia do saldo da conta
10 int getBalance(); // retorna o saldo da conta
11 private:
12 int balance; // membro de dados que armazena o saldo
13 }; // fim da classe Account

```

**Figura M.1** Arquivo de cabeçalho da classe Account.

```

1 // Figura M.2: Account.cpp
2 // Definições de função-membro para a classe Account.
3 #include <iostream>
4 using std::cout;
5 using std::endl;
6
7 #include "Account.h" // inclui a definição da classe Account
8
9 // construtor Account inicializa membro de dados balance
10 Account::Account(int initialBalance)
11 {
12 balance = 0; // assume que balance inicia em 0
13
14 // se initialBalance for maior que 0, configura esse valor como o
15 // saldo [balance] da Account; caso contrário, balance retorna 0
16 if (initialBalance > 0)
17 balance = initialBalance;
18
19 // se initialBalance for negativo, imprime mensagem de erro
20 if (initialBalance < 0)
21 cout << "Error: Initial balance cannot be negative.\n" << endl;
22 } // fim do construtor Account
23
24 // credita (adiciona) uma quantia ao saldo da conta
25 void Account::credit(int amount)
26 {
27 balance = balance + amount; // adiciona amount ao balance
28 } // fim da função credit
29
30 // debita (subtrai) uma quantia do saldo da conta
31 void Account::debit(int amount)
32 {
33 if (amount <= balance) // a quantia de débito não excede o saldo
34 balance = balance - amount;
35
36 else // a quantia de débito excede o saldo
37 cout << "Debit amount exceeded account balance.\n" << endl;
38 } // fim da função debit
39
40 // retorna o saldo da conta
41 int Account::getBalance()
42 {
43 return balance; // fornece o valor do saldo à função chamadora
44 } // fim da função getBalance

```

**Figura M.2** Definição da classe Account.

```

1 // Figura M.3: figM_03.cpp
2 // Cria e manipula um objeto Account.
3 #include <iostream>
4 using std::cin;
5 using std::cout;
6 using std::endl;
7

```

**Figura M.3** Classe de teste para depuração.

(continua)

```

8 // inclui a definição da classe Account a partir de Account.h
9 #include "Account.h"
10
11 // a função main inicia a execução do programa
12 int main()
13 {
14 Account account1(50); // cria o objeto Account
15
16 // exibe o saldo inicial de cada objeto
17 cout << "account1 balance: $" << account1.getBalance() << endl;
18
19 int withdrawalAmount; // armazena a quantia de depósito lida a partir do usuário
20
21 cout << "\nEnter withdrawal amount for account1: "; // prompt
22 cin >> withdrawalAmount; // obtém a entrada do usuário
23 cout << "\nattempting to subtract " << withdrawalAmount
24 << " from account1 balance\n\n";
25 account1.debit(withdrawalAmount); // tenta subtrair de account1
26
27 // exibe os saldos
28 cout << "account1 balance: $" << account1.getBalance() << endl;
29 return 0; // indica terminação bem-sucedida
30 } // fim do main

```

Figura M.3 Classe de teste para depuração.

(continuação)

Nos passos a seguir, você utilizará pontos de interrupção e vários comandos de depurador para examinar o valor da variável `withdrawalAmount` declarada na linha 19 da Figura M.3.

1. *Compilando o programa para depuração.* O depurador GNU só funciona com os arquivos executáveis compilados com a opção de compilador `-g`, o que gera informações utilizadas pelo depurador para ajudar a depurar seus programas. Compile o programa com a opção de linha de comando `-g` digitando `g++ -g -o figM_03 figM_03.cpp Account.cpp`.
2. *Iniciando o depurador.* Digite `gdb figM_03` (Figura M.4). Esse comando iniciará o depurador do GNU e exibirá o prompt (`gdb`) em que você pode inserir comandos.
3. *Executando um programa no depurador.* Execute o programa pelo depurador digitando `run` (Figura M.5). Se você não configurar nenhum ponto de interrupção antes de executar seu programa no depurador, o programa executará até sua conclusão.
4. *Inserindo pontos de interrupção utilizando o depurador do GNU.* Você configura um ponto de interrupção em uma linha específica do código no seu programa. Os números das linhas utilizados nestes passos são provenientes do código-fonte na Figura M.3. Configure um ponto de interrupção na linha 17 no código-fonte digitando `break 17`. O comando `break` insere um ponto de interrupção no número de linha especificado depois do comando. Você pode configurar quantos pontos de interrupção forem necessários. Cada ponto de interrupção é identificado em termos da ordem em que foi criado. O primeiro ponto de interrupção

```

~/Debug$ gdb figM_03
GNU gdb 6.1-debian
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-linux"...Using host libthread_db library "/lib/libthread_db.so.1".

(gdb)

```

Figura M.4 Iniciando o depurador para executar o programa.



```
(gdb) run
Starting program: /home/student/Debug/figM_03
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance

account1 balance: $37

Program exited normally.
(gdb)
```

**Figura M.5** Executando o programa sem pontos de interrupção configurados.

criado é conhecido como Breakpoint 1. Configure outro ponto de interrupção na linha 25 digitando `break 25` (Figura M.6). Quando o programa executa, ele suspende a execução em qualquer linha que contenha um ponto de interrupção e dizemos que o programa está no **modo de interrupção**. Pontos de interrupção podem ser até mesmo configurados depois de o processo de depuração ter iniciado. [Nota: Se você não tiver uma listagem numerada para seu código, pode utilizar o comando `list` para gerar uma saída do seu código com números de linha. Para informações adicionais sobre o comando `list`, digite `help list` a partir do prompt do gdb.]

5. *Executando o programa e começando o processo de depuração.* Digite `run` para executar seu programa e iniciar o processo de depuração (Figura M.7). O programa pausa quando a execução alcança o ponto de interrupção na linha 17. Nesse ponto, o depurador notifica você de que um ponto de interrupção foi alcançado e exibe o código-fonte dessa linha (17). Essa linha do código contém a próxima instrução que será executada.
6. *Utilizando o comando **continue** para retomar a execução.* Digite `continue`. O comando `continue` faz com que o programa continue a executar até o próximo ponto de interrupção ser alcançado (linha 25). Insira 13 no prompt. O depurador notifica você quando a execução alcança o segundo ponto de interrupção (Figura M.8). Observe que a saída normal de `figM_03` aparece entre as mensagens no depurador.
7. *Examinando o valor de uma variável.* Digite `print withdrawalAmount` para exibir o valor atual armazenado na variável `withdrawalAmount` (Figura M.9). O comando `print` permite-lhe examinar dentro do computador o valor de uma das suas variáveis. Esse comando o ajudará a encontrar e eliminar erros de lógica no seu código. Observe que o valor exibido é 13 — o valor lido e atribuído a `withdrawalAmount` na linha 22 da Figura M.3. Utilize o comando `print` para gerar saída do conteúdo do objeto `account1`. Quando um objeto é enviado para a saída pelo depurador com o comando `print`, ele é enviado com chaves cercando os membros de dados do objeto. Nesse caso, há um único membro de dados — `balance` — que tem valor 50.

```
(gdb) break 17
Breakpoint 1 at 0x80487d8: file figM_03.cpp, line 17.
(gdb) break 25
Breakpoint 2 at 0x8048871: file figM_03.cpp, line 25.
(gdb)
```

**Figura M.6** Configurando dois pontos de interrupção no programa.

```
(gdb) run
Starting program: /home/student/Debug/figM_03

Breakpoint 1, main () at figM_03.cpp:17
17 cout << "account1 balance: $" << account1.getBalance() << endl;
(gdb)
```

**Figura M.7** Executando o programa até alcançar o primeiro ponto de interrupção.

```
(gdb) continue
Continuing.
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance

Breakpoint 2, main () at figM_03.cpp:25
25 account1.debit(withdrawalAmount); // tenta subtrair de account1
(gdb)
```

**Figura M.8** Continuando a execução até o segundo ponto de interrupção ser alcançado.

```
(gdb) print withdrawalAmount
$1 = 13
(gdb) print account1
$2 = {balance = 50}
(gdb)
```

**Figura M.9** Imprimindo os valores das variáveis.

8. *Utilizando variáveis de conveniência.* Quando o comando `print` é utilizado, o resultado é armazenado em uma variável de conveniência como `$1`. Variáveis de conveniência, que são variáveis temporárias, identificadas utilizando um sinal de cifrão seguido por um inteiro, são criadas pelo depurador quando você imprime valores durante sua sessão de depuração. Uma variável de conveniência pode ser utilizada no processo de depuração para realizar aritmética e avaliar expressões booleanas. Digite `print $1`. O depurador exibe o valor de `$1` (Figura M.10), que contém o valor de `withdrawalAmount`. Observe que a impressão do valor `$1` cria uma nova variável de conveniência — `$3`.
9. *Continuando a execução.* Digite `continue` para continuar a execução do programa. O depurador não encontra nenhum ponto de interrupção adicional, então ele continua executar e por fim termina (Figura M.11).
10. *Removendo um ponto de interrupção.* Você pode exibir uma lista de todos os pontos de interrupção no programa digitando `info break`. Para remover um ponto de interrupção, digite `delete`, seguido por um espaço e o número do ponto de interrupção a remover. Remova o primeiro ponto de interrupção digitando `delete 1`. Remova o segundo ponto de interrupção também.

```
(gdb) print $1
$3 = 13
(gdb)
```

**Figura M.10** Imprimindo uma variável de conveniência.

```
(gdb) continue
Continuing.
account1 balance: $37

Program exited normally.
(gdb)
```

**Figura M.11** Finalizando a execução do programa.

Agora digite `info break` para listar os pontos de interrupção restantes no programa. O depurador deve indicar que nenhum ponto de interrupção está configurado (Figura M.12).

11. *Executando o programa sem pontos de interrupção.* Digite `run` para executar o programa. Insira o valor 13 no prompt. Como você removeu com sucesso os dois pontos de interrupção, a saída do programa é exibida sem o depurador entrar no modo de interrupção (Figura M.13).
12. *Utilizando o comando **quit**.* Utilize o **comando quit** para encerrar a sessão de depuração (Figura M.14). Esse comando faz com que o depurador termine.

Nesta seção, você aprendeu a ativar o depurador utilizando o comando `gdb` e a executar um programa com o comando `run`. Você viu como configurar um ponto de interrupção em um número de linha particular na função `main`. O comando `break` também pode ser utilizado para configurar um ponto de interrupção em um número de linha em outro arquivo ou em uma função particular. Digitar `break`, depois o nome de arquivo, dois-pontos e o número da linha configurará um ponto de interrupção em uma linha em outro arquivo. Digitar `break` e depois um nome de função fará com que o depurador entre no modo de interrupção sempre que essa função for chamada.

Também nesta seção, você viu como o comando `help list` fornecerá informações adicionais sobre o comando `list`. Se você tiver qualquer dúvida sobre o depurador ou qualquer de seus comandos, digite `help` ou `help` seguido pelo nome do comando para informações adicionais.

Por fim, você aprendeu a examinar variáveis com o comando `print` e a remover pontos de interrupção com o comando `delete`. Você aprendeu a utilizar o comando `continue` para continuar a execução depois que um ponto de interrupção é alcançado e o comando `quit` para finalizar o depurador.

```
(gdb) info break
Num Type Disp Enb Address What
1 breakpoint keep y 0x080487d8 in main at figM_03.cpp:17
 breakpoint already hit 1 time
2 breakpoint keep y 0x08048871 in main at figM_03.cpp:25
 breakpoint already hit 1 time
(gdb) delete 1
(gdb) delete 2
(gdb) info break
No breakpoints or watchpoints.
(gdb)
```

**Figura M.12** Visualizando e removendo pontos de interrupção.

```
(gdb) run
Starting program: /home/student/Debug/figM_03
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance

account1 balance: $37

Program exited normally.
(gdb)
```

**Figura M.13** Programa executando sem pontos de interrupção configurados.

```
(gdb) quit
~/Debug$
```

**Figura M.14** Saindo do depurador utilizando o comando `quit`.

### M.3 Os comandos print e set

Na seção anterior, você aprendeu a utilizar o comando `print` do depurador para examinar o valor de uma variável durante a execução do programa. Nesta seção, você aprenderá a utilizar o comando `print` para examinar o valor de expressões mais complexas. Você também aprenderá a utilizar o **comando set**, que permite ao programador atribuir novos valores a variáveis. Supomos que você esteja trabalhando no diretório que contém os exemplos deste apêndice e que tenha compilado para depuração com a opção de compilador `-g`.

1. *Iniciando a depuração.* Digite `gdb figM_03` para iniciar o depurador do GNU.
2. *Inserindo um ponto de interrupção.* Configure um ponto de interrupção na linha 25 no código-fonte digitando `break 25` (Figura M.15).
3. *Executando o programa e alcançando um ponto de interrupção.* Digite `run` para iniciar o processo de depuração (Figura M.16). Isso fará com que `main` execute até o ponto de interrupção na linha 25 ser alcançado. Isso suspende a execução do programa e alterna o programa para o modo de interrupção. A instrução na linha 25 é a próxima instrução que será executada.
4. *Avaliando expressões aritméticas e booleanas.* Lembre-se, a partir do foi discutido na Seção M.2, de que, depois que o programa entra no modo de interrupção, você pode explorar os valores das variáveis do programa utilizando o comando `print` do depurador. Você também pode utilizar o comando `print` para avaliar expressões aritméticas e booleanas. Digite `print withdrawalAmount - 2`. Observe que o comando `print` retorna o valor 11 (Figura M.17). Entretanto, esse comando na verdade não altera o valor de `withdrawalAmount`. Digite `print withdrawalAmount == 11`. As expressões que contêm o símbolo `==` são tratadas como expressões booleanas. O valor retornado é `false` (Figura M.17) porque `withdrawalAmount` atualmente não contém o valor 11 — `withdrawalAmount` ainda é 13.
5. *Modificando valores.* O depurador permite alterar os valores das variáveis durante a execução do programa. Isso pode ser valioso para experimentar diferentes valores e localizar erros de lógica nos programas. Você pode utilizar o comando `set` do depurador

```
(gdb) break 25
Breakpoint 1 at 0x8048871: file figM_03.cpp, line 25.
(gdb)
```

**Figura M.15** Configurando um ponto de interrupção no programa.

```
(gdb) run
Starting program: /home/student/Debug/figM_03
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance

Breakpoint 1, main () at figM_03.cpp:25
25 account1.debit(withdrawalAmount); // tenta subtrair de account1
(gdb)
```

**Figura M.16** Executando o programa até o ponto de interrupção na linha 25 ser alcançado.

```
(gdb) print withdrawalAmount - 2
$1 = 11
(gdb) print withdrawalAmount == 11
$2 = false
(gdb)
```

**Figura M.17** Imprimindo expressões com o depurador.

para alterar o valor de uma variável. Digite `set withdrawalAmount = 42`. O depurador altera o valor de `withdrawalAmount`. Digite `print withdrawalAmount` para exibir seu novo valor (Figura M.18).

6. *Visualizando o resultado do programa.* Digite `continue` para continuar a execução do programa. A linha 25 da Figura M.3 executa, passando `withdrawalAmount` para a função-membro `Account debit`. A função `main` então exibe o novo saldo. Observe que o resultado é \$8 (Figura M.19). Isso mostra que o passo anterior alterou o valor de `withdrawalAmount` a partir do seu valor inicial (13) para 42.
7. *Utilizando o comando **quit**.* Utilize o comando `quit` para finalizar a sessão de depuração (Figura M.20). Esse comando faz com que o depurador termine.

Nesta seção, você aprendeu a utilizar o comando `print` do depurador para avaliar expressões aritméticas e booleanas. Você também aprendeu a utilizar o comando `set` para modificar o valor de uma variável durante a execução do seu programa.

## M.4 Controlando a execução utilizando os comandos `step`, `finish` e `next`

Às vezes você precisará executar um programa linha por linha para localizar e corrigir erros. Inspecionar passo a passo uma parte do seu programa dessa maneira pode ajudá-lo a verificar se o código de uma função executa corretamente. Nesta seção, você aprenderá a utilizar o depurador para essa tarefa. Os comandos que você aprende aqui lhe permitem executar uma função linha por linha, executar todas as instruções de uma função de uma vez ou executar somente as instruções restantes de uma função (se você já executou algumas instruções dentro da função). Mais uma vez, supomos que você esteja trabalhando no diretório que contém os exemplos deste apêndice e que tenha compilado para depuração com a opção de compilador `-g`.

1. *Iniciando o depurador.* Inicie o depurador digitando `gdb figM_03`.
2. *Configurando um ponto de interrupção.* Digite `break 25` para configurar um ponto de interrupção na linha 25.
3. *Executando o programa.* Execute o programa digitando `run`. Depois que o programa exibe suas duas mensagens de saída, o depurador indica que o ponto de interrupção foi alcançado e exibe o código na linha 25. O depurador e o programa então pausam e esperam o próximo comando ser inserido.
4. *Utilizando o comando **step**.* O comando `step` executa a próxima instrução no programa. Se a próxima instrução a executar for uma chamada de função, o controle é transferido para a função chamada. O comando `step` permite entrar em uma função e estudar as instruções individuais dessa função. Por exemplo, você pode utilizar os comandos `print` e `set` para visualizar e modificar as variáveis dentro da função. Digite `step` para entrar na função-membro `debit` da classe `Account` (Figura M.2).

```
(gdb) set withdrawalAmount = 42
(gdb) print withdrawalAmount
$3 = 42
(gdb)
```

**Figura M.18** Configurando o valor de uma variável enquanto no modo de interrupção.

```
(gdb) continue
Continuing.
account1 balance: $8

Program exited normally.
(gdb)
```

**Figura M.19** Utilizando uma variável modificada na execução de um programa.

```
(gdb) quit
~/Debug$
```

**Figura M.20** Saindo do depurador utilizando o comando `quit`.

O depurador indica que o passo foi completado e exibe a próxima instrução executável (Figura M.21) — nesse caso, a linha 33 da classe `Account` (Figura M.2).

5. *Utilizando o comando **finish**.* Depois que você iniciou a inspeção passo a passo (*step into*) da função-membro `debit`, digite **finish**. Esse comando executa as instruções remanescentes na função e retorna o controle ao local em que a função foi chamada. O comando `finish` executa as instruções restantes na função-membro `debit`, e então pausa na linha 28 em `main` (Figura M.22). Em funções longas, talvez você queira examinar algumas linhas-chave do código e então continuar a depurar o código do chamador. O comando `finish` é útil para situações em que você não quer continuar a inspeção (*stepping through*) da função inteira linha por linha.
6. *Utilizando o comando **continue** para continuar a execução.* Insira o comando `continue` para continuar a execução. Nenhum ponto de interrupção adicional é alcançado, então o programa termina.
7. *Executando o programa novamente.* Os pontos de interrupção persistem até o fim da sessão de depuração em que eles são configurados — mesmo depois da execução do programa, todos os pontos de interrupção são mantidos. O ponto de interrupção que você configura no Passo 2 estará aí na próxima execução do programa. Digite `run` para executar o programa. Como no Passo 3, o programa executa até o ponto de interrupção na linha 25 ser alcançado, então o depurador pausa e espera o próximo comando (Figura M.23).
8. *Utilizando o comando **next**.* Digite **next**. Esse comando comporta-se como o comando `step`, exceto quando a próxima instrução a executar contém uma chamada de função. Nesse caso, a função chamada executa em sua totalidade e o programa avança até a próxima linha executável depois da chamada de função (Figura M.24). No Passo 4, o comando `step` entra na função

```
(gdb) step
Account::debit (this=0xbffffd70, amount=13) at Account.cpp:33
33 if (amount <= balance) // debit amount does not exceed balance
(gdb)
```

**Figura M.21** Utilizando o comando `step` para entrar em uma função.

```
(gdb) finish
Run till exit from #0 Account::debit (this=0xbffffd70, amount=13)
 at Account.cpp:33
main () at figM_03.cpp:28
28 cout << "account1 balance: $" << account1.getBalance() << endl;
(gdb)
```

**Figura M.22** Utilizando o comando `finish` para concluir a execução de uma função e retornar à função chamadora.

```
(gdb) run
Starting program: /home/student/Debug/figM_03
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance

Breakpoint 1, main () at figM_03.cpp:25
25 account1.debit(withdrawalAmount); // try to subtract from account1
(gdb)
```

**Figura M.23** Reiniciando o programa.

chamada. Nesse exemplo, o comando `next` faz com que a função-membro `Account::debit` execute, e então o depurador pausa na linha 28.

9. *Utilizando o comando **quit**.* Utilize o comando `quit` para finalizar a sessão de depuração (Figura M.25). Enquanto o programa está executando, esse comando faz com que o programa imediatamente termine em vez de executar as instruções restantes em `main`.

Nesta seção, você aprendeu a utilizar os comandos `step` e `finish` do depurador para depurar chamadas de função durante a execução do seu programa. Você viu como o comando `next` pode ser utilizado para a inspeção pular (*step over*) uma chamada de função. Você também aprendeu que o comando `quit` encerra uma sessão de depuração.

## M.5 O comando watch

Nesta seção, apresentamos o **comando watch**, que instrui o depurador a monitorar um membro de dados. Quando esse membro de dados estiver prestes a mudar, o depurador notificará você. Nesta seção, você aprenderá a utilizar o comando `watch` para ver como o membro de dados `balance` do objeto `Account` é modificado durante a execução do programa.

1. *Iniciando o depurador.* Inicie o depurador digitando `gdb figM_03`.
2. *Executando o programa.* Digite `break 14` para configurar um ponto de interrupção na linha 14. Execute o programa com o comando `run`. O depurador e o programa pausarão no ponto de interrupção na linha 14 (Figura M.26).
3. *Observando um membro de dados da classe.* Configure um ponto de observação (*watch*) no membro de dados `balance` de `account1` digitando `watch account1.balance` (Figura M.27). Esse ponto de observação é rotulado como `watchpoint 2` porque os pontos de observação são rotulados com os mesmos números que os pontos de interrupção. Você pode configurar um ponto de observação em qualquer variável ou membro de dados de um objeto atualmente em escopo durante a execução

```
(gdb) next
28 cout << "account1 balance: $" << account1.getBalance() << endl;
(gdb)
```

**Figura M.24** Utilizando o comando `next` para executar uma função em sua totalidade.

```
(gdb) quit
The program is running. Exit anyway? (y or n) y
~/Debug$
```

**Figura M.25** Saindo do depurador utilizando o comando `quit`.

```
(gdb) run
Starting program: /home/student/Debug/figM_03

Breakpoint 1, main () at figM_03.cpp:14
14 Account account1(50); // create Account object
(gdb)
```

**Figura M.26** Executando o programa até o primeiro ponto de interrupção.

```
(gdb) watch account1.balance
Hardware watchpoint 2: account1.balance
(gdb)
```

**Figura M.27** Configurando um ponto de observação em um membro de dados.



do depurador. Sempre que o valor de uma variável observada mudar, o depurador entra no modo de interrupção e notifica você de que o valor mudou.

4. *Continuando o programa.* Inspeção passo a passo (*step into*) no construtor `Account` com o comando `step`. O depurador exibirá a linha 12 da Figura M.2, que é a primeira linha no construtor. Utilize o comando `step` novamente para executar essa linha de código. O depurador agora o notificará de que o valor do membro de dados `balance` mudará (Figura M.28). Quando o programa inicia, uma instância de `Account` é criada. Quando o construtor para esse objeto executa, o valor 0 é primeiro atribuído ao membro de dados `balance`. O depurador o notifica de que o valor de `balance` foi alterado.
5. *Terminando o construtor.* Digite `step` para executar a linha 16, então digite `step` novamente para executar a linha 17. O depurador o notificará de que valor do membro de dados `balance` mudou de 0 para 50 (Figura M.29).
6. *Retirando dinheiro da conta.* Digite `continue` para continuar a execução e insira um valor de retirada no prompt. O programa executa normalmente. A linha 25 da Figura M.3 chama a função-membro `Account::debit` para subtrair do objeto `Account` `balance` um `amount` especificado. A linha 34 da Figura M.2 dentro da função `debit` altera o valor do saldo. O depurador o notifica dessa alteração e entra no modo de interrupção (Figura M.30).
7. *Continuando a execução.* Digite `continue` — o programa terminará de executar a função `main` porque o programa não tenta nenhuma alteração adicional em `balance`. O depurador remove o ponto de observação no membro de dados `balance` de `account1` porque a variável sai de escopo quando a função `main` termina. Remover o ponto de observação faz com que o depurador entre no modo de interrupção. Digite `continue` novamente para terminar a execução do programa (Figura M.31).
8. *Reiniciando o depurador e redefinindo o ponto de monitoração sobre a variável.* Digite `run` para reiniciar o depurador. Mais uma vez, configure um ponto de observação no membro de dados `account1::balance` digitando `watch account1.balance`. Esse ponto de observação é rotulado como `watchpoint 3`. Digite `continue` para continuar a execução (Figura M.32).
9. *Removendo o ponto de observação no membro de dados.* Suponha que você queira monitorar um membro de dados por somente parte da execução de um programa. Você pode remover o ponto de observação do depurador na variável `balance` digitando `delete 3` (Figura M.33). Digite `continue` — o programa terminará a execução sem reentrar no modo de interrupção.

```
(gdb) step
Account (this=0xbffffd70, initialBalance=50) at Account.cpp:12
12 balance = 0; // pressupõe que o balanço inicia em 0
(gdb) step
Hardware watchpoint 2: account1.balance

Old value = 1073833120
New value = 0
Account (this=0xbffffd70, initialBalance=50) at Account.cpp:16
16 if (initialBalance > 0)
(gdb)
```

**Figura M.28** Inspeção passo a passo do construtor.

```
(gdb) step
17 balance = initialBalance;
(gdb) step
Hardware watchpoint 2: account1.balance

Old value = 0
New value = 50
Account (this=0xbffffd70, initialBalance=50) at Account.cpp:20
20 if (initialBalance < 0)
(gdb)
```

**Figura M.29** Alcançando a notificação de um ponto de observação.

```

(gdb) continue
Continuing.
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance

Hardware watchpoint 2: account1.balance

Old value = 50
New value = 37
0x08048a01 in Account::debit (this=0xbffffd70, amount=13) at Account.cpp:34
34 balance = balance - amount;
(gdb)

```

**Figura M.30** Entrando no modo de interrupção quando uma variável é alterada.

```

(gdb) continue
Continuing.
end of function
account1 balance: $37

Watchpoint 2 deleted because the program has left the block in
which its expression is valid.
0x4012fa65 in exit () from /lib/libc.so.6
(gdb) continue
Continuing.

Program exited normally.
(gdb)

```

**Figura M.31** Continuando até o fim do programa.

```

(gdb) run
Starting program: /home/student/Debug/figM_03

Breakpoint 1, main () at figM_03.cpp:14
14 Account account1(50); // cria o objeto Account
(gdb) watch account1.balance
Hardware watchpoint 3: account1.balance
(gdb) continue
Continuing.
Hardware watchpoint 3: account1.balance

Old value = 1073833120
New value = 0
Account (this=0xbffffd70, initialBalance=50) at Account.cpp:16
16 if (initialBalance > 0)
(gdb)

```

**Figura M.32** Redefinindo o ponto de observação em um membro de dados.

```
(gdb) delete 3
(gdb) continue
Continuing.
account1 balance: $50

Enter withdrawal amount for account1: 13

attempting to subtract 13 from account1 balance

end of function
account1 balance: $37

Program exited normally.
(gdb)
```

**Figura M.33** Removendo um ponto de observação.

Nesta seção, você aprendeu a utilizar o comando `watch` para permitir que o depurador o notifique de alterações no valor de um membro de dados por toda a vida de um programa. Você também aprendeu a utilizar o comando `delete` para remover um ponto de observação em um membro de dados antes do fim do programa.

## M.6 Síntese

Neste apêndice, você aprendeu a inserir e remover pontos de interrupção no depurador. Os pontos de interrupção permitem pausar a execução do programa para você poder examinar os valores das variáveis com o comando `print` do depurador. Essa capacidade o ajudará a localizar e corrigir erros de lógica nos seus programas. Você viu como utilizar o comando `print` para examinar o valor de uma expressão, e aprendeu a utilizar o comando `set` para alterar o valor de uma variável. Você também aprendeu comandos de depurador (incluindo os comandos `step`, `finish` e `next`) que podem ser utilizados para determinar se uma função está executando corretamente. Você aprendeu a utilizar o comando `watch` para monitorar um membro de dados por todo o escopo desse membro de dados. Por fim, aprendeu a utilizar o comando `info break` para listar todos os pontos de interrupção e pontos de observação configurados para um programa e o comando `delete` para remover pontos de interrupção e pontos de observação individuais.

### Resumo

- O GNU inclui um software chamado depurador, que permite que você monitore a execução de seus programas para localizar e remover erros de lógica.
- O depurador GNU só funciona com os arquivos executáveis compilados com a opção de compilador `-g`, o que gera informações utilizadas pelo depurador para ajudar a depurar seus programas.
- O comando `gdb` irá iniciar o depurador do GNU e permitir que você utilize seus recursos. O comando `run` executará um programa através do depurador.
- Os pontos de interrupção são marcadores que podem ser configurados em qualquer linha executável do código. Quando a execução do programa alcança um ponto de interrupção, a execução pausa.
- O comando `break` insere um ponto de interrupção no número de linha especificado depois do comando.
- Quando o programa executa, ele suspende a execução em qualquer linha que contenha um ponto de interrupção e dizemos que ele está no modo de interrupção.
- O comando `continue` faz com que o programa continue executando até o próximo ponto de interrupção ser alcançado.
- O comando `print` permite que você examine dentro do computador o valor de uma das suas variáveis.
- Quando o comando `print` é utilizado, o resultado é armazenado em uma variável de conveniência como `$1`. Variáveis de conveniência são variáveis temporárias que podem ser utilizadas no processo de depuração para realizar aritmética e avaliar expressões booleanas.
- Você pode exibir uma lista de todos os pontos de interrupção no programa digitando `info break`.
- Para remover um ponto de interrupção, digite `delete`, seguido por um espaço e pelo número do ponto de interrupção a remover.
- Utilize o comando `quit` para finalizar a sessão de depuração.

- O comando `set` permite que o programador atribua novos valores a variáveis.
- O comando `step` executa a próxima instrução no programa. Se a próxima instrução a executar for uma chamada de função, o controle é transferido para a função chamada. O comando `step` permite que você entre em uma função e estude as instruções individuais dessa função.
- O comando `finish` executa as instruções restantes na função e retorna o controle para o lugar onde a função foi chamada.
- O comando `next` comporta-se como o comando `step`, exceto quando a próxima instrução a executar contém uma chamada de função. Nesse caso, a função chamada executa em sua totalidade e o programa avança para a próxima linha executável após a chamada de função.
- O comando `watch` configura um ponto de observação em qualquer variável ou membro de dados de um objeto atualmente em escopo durante execução do depurador. Sempre que o valor de uma variável observada mudar, o depurador entra no modo de interrupção e notifica você de que o valor mudou.

## Terminologia

<code>break</code> , comando	<code>gdb</code> , comando	<code>quit</code> , comando
<code>continue</code> , comando	<code>info break</code> , comando	<code>run</code> , comando
<code>delete</code> , comando	modo de interrupção	<code>set</code> , comando
depurador	<code>next</code> , comando	<code>step</code> , comando
<code>finish</code> , comando	ponto de interrupção	<code>watch</code> , comando
<code>-g</code> , opção de compilador	<code>print</code> , comando	

## Exercícios de revisão

- M.1** Preencha as lacunas em cada uma das seguintes sentenças:
- Um ponto de interrupção não pode ser configurado em uma \_\_\_\_\_.
  - Você pode examinar o valor de uma expressão utilizando o comando \_\_\_\_\_ do depurador.
  - Você pode modificar o valor de uma variável utilizando o comando \_\_\_\_\_ do depurador.
  - Durante a depuração, o comando \_\_\_\_\_ executa as instruções restantes na função atual e retorna o controle do programa ao local em que a função foi chamada.
  - O comando \_\_\_\_\_ do depurador comporta-se como o comando `step` quando a próxima instrução a executar não contém uma chamada de função.
  - O comando `watch` do depurador permite visualizar todas as alterações em um \_\_\_\_\_.
- M.2** Determine se cada uma das seguintes sentenças é *verdadeira* ou *falsa*. Se *falsa*, explique por quê.
- Quando a execução do programa é suspensa em um ponto de interrupção, a próxima instrução a ser executada é a instrução depois do ponto de interrupção.
  - Pontos de observação podem ser removidos com o comando `remove` do depurador.
  - A opção de compilador `-g` deve ser utilizada ao compilar programas para depuração.

## Respostas dos exercícios de revisão

- M.1** a) linha não executável. b) `print`. c) `set`. d) `finish`. e) `next`. f) membro de dados.
- M.2** a) Falsa. Quando a execução do programa é suspensa em um ponto de interrupção, a próxima instrução a ser executada é a instrução no ponto de interrupção. b) Falsa. Os pontos de observação podem ser removidos utilizando o comando `delete` do depurador. c) Verdadeira.



---

# Bibliografia

---

- Alhir, S. *UML in a nutshell*. Cambridge, MA: O'Reilly & Associates, Inc., 1998.
- Allison, C. "Text processing I", *The C Users Journal*, vol. 10, no. 10, out. 1992, p. 23–28.
- . "Text processing II", *The C Users Journal*, vol. 10, no. 12, dez. 1992, p. 73–77.
- . "Code capsules: a C++ date class, part I", *The C Users Journal*, vol. 11, no. 2, fev. 1993, 123–131.
- . "Conversions and casts", *The C/C++ Users Journal*, vol. 12, no. 9, set. 1994, p. 67–85.
- Almarode, J. "Object security", *Smalltalk Report*, vol. 5, no. 3, nov./dez. 1995, p. 15–17.
- American National Standard, *Programming Language C++*. (ANSI Document ISO/IEC 14882), Nova York, NY: American National Standards Institute, 1998.
- Anderson, A. E. e Heinze, W. J. *C++ programming and fundamental concepts*. Englewood Cliffs, NJ: Prentice Hall, 1992.
- Baker, L. *C mathematical function handbook*. Nova York, NY: McGraw Hill, 1992.
- Bar-David, T. *Object-oriented design for C++*. Englewood Cliffs, NJ: Prentice Hall, 1993.
- Beck, K. "Birds, bees, and browsers—obvious sources of objects", *The Smalltalk Report*, vol. 3, no. 8, jun. 1994, p. 13.
- Becker, P. "Shrinking the big switch statement", *Windows Tech Journal*, vol. 2, no. 5, mai. 1993, p. 26–33.
- Becker, P. "Conversion confusion", *C++ Report*, out. 1993, p. 26–28.
- Berard, E. V. *Essays on object-oriented software engineering, vol. I*. Englewood Cliffs, NJ: Prentice Hall, 1993.
- Binder, R. V. "State-based testing", *Object Magazine*, vol. 5, no. 4, ago. 1995, p. 75–78.
- . "State-based testing: sneak paths and conditional transitions", *Object Magazine*, vol. 5, no. 6, out. 1995, p. 87–89.
- Blum, A. *Neural networks in C++: an object-oriented framework for building connectionist systems*. Nova York, NY: John Wiley & Sons, 1992.
- Booch, G. *Object solutions: managing the object-oriented project*. Reading, MA: Addison-Wesley, 1996.
- . *Object-oriented analysis and design with applications, 3 ed.* Reading, MA: Addison-Wesley, 2005.
- ; J. Rumbaugh e I. Jacobson, *The Unified Modeling Language user guide*. Reading, MA: Addison-Wesley, 1999.
- Cargill, T. *C++ programming style*. Reading, MA: Addison-Wesley, 1993.
- Carroll, M. D. e Ellis, M. A. *Designing and coding reusable C++*. Reading, MA: Addison-Wesley, 1995.
- Coplien, J. O. e Schmidt, D. C. *Pattern languages of program design*. Reading, MA: Addison-Wesley, 1995.
- Deitel, H. M. e Deitel, P. J. *Java how to program, 6 ed.* Upper Saddle River, NJ: Prentice Hall, 2005.
- . *C how to program, 4 ed.* Upper Saddle River, NJ: Prentice Hall, 2004.
- e Choffnes, D. R. *Operating systems, 3 ed.* Upper Saddle River, NJ: Prentice Hall, 2004.
- Duncan, R. "Inside C++: friend and virtual functions, and multiple inheritance", *PC Magazine*, 15 out. 1991, p. 417–420.
- Ellis, M. A. e Stroustrup, B. *The annotated C++ reference manual*. Reading, MA: Addison-Wesley, 1990.
- Embley, D. W.; Kurtz B. D. e Woodfield, S. N. *Object-oriented systems analysis: a model-driven approach*. Englewood Cliffs, NJ: Yourdon Press, 1992.
- Entsminger, G. e Eckel, B. *The tao of objects: a beginner's guide to object-oriented programming*. Nova York, NY: Wiley Publishing, 1990.
- Firesmith, D. G. e Henderson-Sellers, B. "Clarifying specialized forms of association in UML and OML", *Journal of Object-Oriented Programming*, mai. 1998, p. 47–50.
- Flamig, B. *Practical data structures in C++*. Nova York, NY: John Wiley & Sons, 1993.
- Fowler, M. *UML distilled: a brief guide to the standard Object Modeling Language, 3 ed.* Reading, MA: Addison-Wesley, 2004.
- Gehani, N. e Roome, W. D. *The concurrent C programming language*. Summit, NJ: Silicon Press, 1989.
- Giancola, A. e Baker, L. "Bit arrays with C++", *The C Users Journal*, vol. 10, no. 7, jul. 1992, p. 21–26.
- Glass, G. e Schuchert, B. *The STL <primer>*. Upper Saddle River, NJ: Prentice Hall PTR, 1995.

- Gooch, T. "Obscure C++", *Inside Microsoft Visual C++*, vol. 6, no. 11, nov. 1995, p. 13–15.
- Hansen, T. L. *The C++ answer book*. Reading, MA: Addison-Wesley, 1990.
- Henricson, M. e Nyquist, E. *Industrial strength C++: rules and recommendations*. Upper Saddle River, NJ: Prentice Hall, 1997.
- International Standard: Programming Language—C++*. ISO/IEC 14882:1998. Nova York, NY: American National Standards Institute, 1998.
- Jacobson, I. "Is object technology software's industrial platform?", *IEEE Software Magazine*, vol. 10, no. 1, jan. 1993, p. 24–30.
- Jaeschke, R. *Portability and the C language*. Indianapolis, IN: Sams Publishing, 1989.
- Johnson, L. J. "Model behavior", *Enterprise Development*, mai. 2000, p. 20–28.
- Josuttis, N. *The C++ Standard Library: a tutorial and reference*. Boston, MA: Addison-Wesley, 1999.
- Knight, A. "Encapsulation and information hiding", *The Smalltalk Report*, vol. 1, no. 8, jun. 1992, p. 19–20.
- Koenig, A. "What is C++ anyway?", *Journal of Object-Oriented Programming*, abr./mai. 1991, p. 48–52.
- . "Implicit base class conversions", *The C++ Report*, vol. 6, no. 5, jun. 1994, p. 18–19.
- e Stroustrup, B. "Exception handling for C++ (revised)", *Proceedings of the USENIX C++ Conference*, San Francisco, CA, abr. 1990.
- e Moo, B. *Ruminations on C++: a decade of programming insight and experience*. Reading, MA: Addison-Wesley, 1997.
- Krusee, R. L. e Ryba, A. J. *Data structures and program design in C++*. Upper Saddle River, NJ: Prentice Hall, 1999.
- Langer, A. e Kreft, K. *Standard C++ IOStreams and locales: advanced programmer's guide and reference*. Reading, MA: Addison-Wesley, 2000.
- Lejter, M.; Meyers S. e Reiss, S. P. "Support for maintaining object-oriented programs", *IEEE Transactions on Software Engineering*, vol. 18, no. 12, dez. 1992, p. 1045–1052.
- Lippman, S. B. e Lajoie, J. *C++ primer, 3 ed.*, Reading, MA: Addison-Wesley, 1998.
- Lorenz, M. *Object-oriented software development: a practical guide*. Englewood Cliffs, NJ: Prentice Hall, 1993.
- . "A brief look at inheritance metrics", *The Smalltalk Report*, vol. 3, no. 8, jun. 1994, p. 1, 4–5.
- Martin, J. *Principles of object-oriented analysis and design*. Englewood Cliffs, NJ: Prentice Hall, 1993.
- Martin, R. C. *Designing object-oriented c++ applications using the booch method*. Englewood Cliffs, NJ: Prentice Hall, 1995.
- Matsche, J. J. "Object-oriented programming in standard C", *Object Magazine*, vol. 2, no. 5, jan./fev. 1993, p. 71–74.
- McCabe, T. J. e Watson, A. H. "Combining comprehension and testing in object-oriented development", *Object Magazine*, vol. 4, no. 1, mar./abr. 1994, p. 63–66.
- McLaughlin, M. e Moore, A. "Real-time extensions to the UML", *Dr. Dobb's Journal*, dez. 1998, p. 82–93.
- Melewska, D. "UML gains ground", *Application development trends*, out. 1998, p. 34–44.
- . "UML: ready for prime time?" *Application Development Trends*, nov. 1997, p. 30–44.
- . "Wherefore and what now, UML?" *Application Development Trends*, dez. 1999, p. 61–68.
- Meyer, B. *Object-oriented software construction, 2 ed.* Englewood Cliffs, NJ: Prentice Hall, 1997.
- . *Eiffel: the language*. Englewood Cliffs, NJ: Prentice Hall, 1992.
- e Mandrioli, D. *Advances in object-oriented software engineering*. Englewood Cliffs, NJ: Prentice Hall, 1992.
- Meyers, S. "Mastering user-defined conversion functions", *The C/C++ Users Journal*, vol. 13, no. 8, ago. 1995, p. 57–63.
- . *More effective C++: 35 new ways to improve your programs and designs*. Reading, MA: Addison-Wesley, 1996.
- . *Effective C++: 50 specific ways to improve your programs and designs, 2 ed.* Reading, MA: Addison-Wesley, 1998.
- . *Effective STL: 50 specific ways to improve your use of the Standard Template Library*. Reading, MA: Addison-Wesley, 2001.
- Muller, P. *Instant UML*. Birmingham, UK: Wrox Press Ltd, 1997.
- Murray, R. *C++ strategies and tactics*. Reading, MA: Addison-Wesley, 1993.
- Musser, D. R. e Stepanov, A. A. "Algorithm-oriented generic libraries", *Software Practice and Experience*, vol. 24, no. 7, jul. 1994.
- ; Derge, G. J. e Saini, A. *STL tutorial and reference guide: C++ programming with the Standard Template Library, 2 ed.* Reading, MA: Addison-Wesley, 2001.
- Nerson, J. M. "Applying object-oriented analysis and design", *Communications of the ACM*, vol. 35, no. 9, set. 1992, p. 63–74.



- Nierstrasz, O.; Gibbs, S. e Tschritzis, D. “Component-oriented software development”, *Communications of the ACM*, vol. 35, no. 9, set. 1992, p. 160–165.
- Perry, P. “UML steps to the plate”, *Application Development Trends*, mai. 1999, p. 33–36.
- Pinson, L. J. e Wiener, R. S. *Applications of object-oriented programming*. Reading, MA: Addison-Wesley, 1990.
- Pittman, M. “Lessons learned in managing object-oriented development”, *IEEE Software Magazine*, vol. 10, no. 1, jan. 1993, p. 43–53.
- Plauger, P. J. *The Standard C Library*. Englewood Cliffs, NJ: Prentice Hall, 1992.
- Plauger, D. “Making C++ safe for threads”, *The C Users Journal*, vol. 11, no. 2, fev. 1993, p. 58–62.
- Pohl, I. *C++ distilled: a concise ANSI/ISO reference and style guide*. Reading, MA: Addison-Wesley, 1997.
- Press, W. H.; Teukolsky, S. A.; Vetterling, W. T. e Flannery, B. P. *Numerical recipes in C: the art of scientific computing*. Cambridge, MA: Cambridge University Press, 1992.
- Prieto-Diaz, R. “Status report: software reusability”, *IEEE Software*, vol. 10, no. 3, mai. 1993, p. 61–66.
- Prince, T. “Tuning up math functions”, *The C Users Journal*, vol. 10, no. 12, dez. 1992.
- Prosis, J. “Wake up and smell the MFC: using the Visual C++ classes and applications framework”, *Microsoft Systems Journal*, vol. 10, no. 6, jun. 1995, p. 17–34.
- Rabinowitz, H. e Schaap, C. *Portable C*. Englewood Cliffs, NJ: Prentice Hall, 1990.
- Reed, D. R. “Moving from C to C++”, *Object Magazine*, vol. 1, no. 3, set./out. 1991, p. 46–60.
- Ritchie, D. M. “The UNIX system: the evolution of the UNIX time-sharing system”, *AT&T Bell Laboratories Technical Journal*, vol. 63, no. 8, parte 2, out. 1984, p. 1577–1593.
- ; Johnson, S. C.; Lesk, M. E. e Kernighan, B. W. “UNIX time-sharing system: the C programming language”, *The Bell System Technical Journal*, vol. 57, no. 6, parte 2, jul./ago. 1978, p. 1991–2019.
- Rosler, L. “The UNIX system: the evolution of C—past and future”, *AT&T Laboratories Technical Journal*, vol. 63, no. 8, parte 2, out. 1984, p. 1685–1699.
- Robson, R. *Using the STL: The C++ Standard Template Library*. Nova York, NY: Springer Verlag, 2000.
- Rubin, K. S. e Goldberg, A. “Object behavior analysis”, *Communications of the ACM*, vol. 35, no. 9, set. 1992, p. 48–62.
- Rumbaugh, J.; Blaha, M.; Premerlani, W.; Eddy, F. e Lorensen, W. *Object-oriented modeling and design*. Englewood Cliffs, NJ: Prentice Hall, 1991.
- ; Jacobson, I. e Booch, G. *The Unified Modeling Language reference manual, 2 ed.* Reading, MA: Addison-Wesley, 2005.
- Saks, D. “Inheritance”, *The C Users Journal*, mai. 1993, p. 81–89.
- Schildt, H. *STL programming from the ground up*. Berkeley, CA: Osborne McGraw-Hill, 1999.
- Schlaer, S. e Mellor, S. J. *Object lifecycles: modeling the world in states*. Englewood Cliffs, NJ: Prentice Hall, 1992.
- Sedgwick, R. *Bundle of algorithms in C++, parts 1–5: fundamentals, data structures, sorting, searching, and graph algorithms (3 ed.)*. Reading, MA: Addison-Wesley, 2002.
- Sessions, R. *Class construction in C and C++: object-oriented programming*. Englewood Cliffs, NJ: Prentice Hall, 1992.
- Skelly, C. “Pointer power in C and C++”, *The C Users Journal*, vol. 11, no. 2, fev. 1993, p. 93–98.
- Snyder, A. “The essence of objects: concepts and terms”, *IEEE Software Magazine*, vol. 10, no. 1, jan. 1993, p. 31–42.
- Stepanov, A. e Lee, M. “The Standard Template Library”, 31 out. 1995 <[www.cs.rpi.edu/~musser/doc.ps](http://www.cs.rpi.edu/~musser/doc.ps)>.
- Stroustrup, B. “The UNIX system: data abstraction in C”, *AT&T Bell Laboratories Technical Journal*, vol. 63, no. 8, parte 2, out. 1984, p. 1701–1732.
- . “What is object-oriented programming?” *IEEE Software*, vol. 5, no. 3, mai. 1988, p. 10–20.
- . “Parameterized types for C++”, *Proceedings of the USENIX C++ Conference*, Denver, CO, out. 1988.
- . “Why consider language extensions?: maintaining a delicate balance”, *The C++ Report*, set. 1993, p. 44–51.
- . “Making a vector fit for a standard”, *The C++ Report*, out. 1994.
- . *The design and evolution of C++*. Reading, MA: Addison-Wesley, 1994.
- . *The C++ programming language, special third edition*. Reading, MA: Addison-Wesley, 2000.
- Taligent’s guide to designing programs: well-mannered object-oriented design in C++. Reading, MA: Addison-Wesley, 1994.
- Taylor, D. *Object-oriented information systems: planning and implementation*. Nova York, NY: John Wiley & Sons, 1992.

- Tondo, C. L. e Gimpel, S. E. *The C answer book*. Englewood Cliffs, NJ: Prentice Hall, 1989.
- Urlocker, Z. "Polymorphism unbounded", *Windows Tech Journal*, vol. 1, no. 1, jan. 1992, p. 11–16.
- Van Camp, K. E. "Dynamic inheritance using filter classes", *The C/C++ Users Journal*, vol. 13, no. 6, jun. 1995, p. 69–78.
- Vilot, M. J. "An introduction to the Standard Template Library", *The C++ Report*, vol. 6, no. 8, out. 1994.
- Voss, G. *Object-oriented programming: an introduction*. Berkeley, CA: Osborne McGraw-Hill, 1991.
- . "Objects and messages", *Windows Tech Journal*, fev. 1993, p. 15–16.
- Wang, B. L. e Wang, J. "Is a deep class hierarchy considered harmful?", *Object Magazine*, vol. 4, no. 7, nov./dez. 1994, p. 35–36.
- Weisfeld, M. "An alternative to large switch statements", *The C Users Journal*, vol. 12, no. 4, abr. 1994, p. 67–76.
- Weiskamp, K. e Flamig, B. *The complete C++ primer, 2 ed.* Orlando, FL: Academic Press, 1993.
- Wiebel, M. e Halladay, S. "Using OOP techniques instead of *switch* in C++", *The C Users Journal*, vol. 10, no. 10, out. 1993, p. 105–112.
- Wilde, N. e Huitt, R. "Maintenance support for object-oriented programs", *IEEE Transactions on Software Engineering*, vol. 18, no. 12, dez. 1992, p. 1038–1044.
- Wilde, N.; Matthews, P. e Huitt, R. "Maintaining object-oriented software", *IEEE Software Magazine*, vol. 10, no. 1, jan. 1993, p. 75–80.
- Wilson, G. V. e Lu, P. *Parallel programming using C++*. Cambridge, MA: MIT Press, 1996.
- Wilt, N. "Templates in C++", *The C Users Journal*, mai. 1993, p. 33–51.
- Wirfs-Brock, R.; Wilkerson, B. e Wiener, L. *Designing object-oriented software*. Englewood Cliffs, NJ: Prentice Hall PTR, 1990.
- Wyatt, B. B.; Kavi, K. e Hufnagel, S. "Parallelism in object-oriented languages: a survey", *IEEE Software*, vol. 9, no. 7, nov. 1992, p. 56–66.
- Yamazaki, S.; Kajihara, K.; Ito, M. e Yasuhara, R. "Object-oriented design of telecommunication software", *IEEE Software Magazine*, vol. 10, no. 1, jan. 1993, p. 81–87.

# Índice remissivo

[Nota: As referências de páginas para ocorrências de definições dos termos aparecem em azul.]

## Símbolos

!, operador NÃO lógico, 161, [162](#), 969  
tabela-verdade, 162  
!=, operador de desigualdade, 38, 454, 969  
##, operador de pré-processador, [1025](#)  
#, operador de pré-processador, [1025](#)  
\$, prompt de linha de comando UNIX, 1002  
\$, operador módulo, 35  
% =, operador de atribuição de módulo, 121  
& e \*, como operadores opostos, 314  
& para indicar referência, 210  
em uma lista de parâmetros, 211  
&&, operador E lógico, 161, [161](#), 225, 969  
tabela-verdade, 161, 162  
&, E sobre bits, [853](#), 969  
&, operador de endereço, 313, 315  
&=, operador de atribuição E sobre bits, [860](#), 950, 969  
\*, operador de desreferência ou de indireção, [314](#), 315  
\*, operador de multiplicação, 35  
\*=, operador de atribuição de multiplicação, 121  
--, operador de decremento prefixado, [121](#)  
--, operador de pós-decremento, [121](#)  
,, operador vírgula, [144](#)  
. \*, operador de ponteiro para membro de classe, [972](#), 972, 973  
/, operador de divisão, 35  
//, comentário de uma única linha, [27](#)  
/=, operador de atribuição de divisão, 121  
::, operador de resolução de escopo binário, [76](#), 436  
::, operador unário de resolução de escopo, [215](#), 968, 972  
?:, operador condicional ternário, [100](#), 225  
\", seqüência de escape de caractere de aspas duplas, 29  
\\', seqüência de escape de caractere de aspas simples, 29  
\\, seqüência de escape de caractere de barra invertida, 29  
'\0', caractere nulo, [346](#)  
\\a, seqüência de escape de alerta, 29  
'\n', caractere de nova linha, 346  
\\n, seqüência de escape de nova linha, 29  
\\r, seqüência de escape de retorno de carro, 29  
\\t, seqüência de escape de tabulação, 29  
^, operador OU exclusivo sobre bits, [853](#), 969  
^=, operador de atribuição OU exclusivo sobre bits, [860](#), 950, 969  
|, operador OU inclusivo sobre bits, [853](#), 969  
||, operador OU lógico, 161, [161](#), 225, 969  
tabela-verdade, 161, 162  
|=, operador de atribuição OU inclusivo sobre bits, [860](#), 950, 969  
~, operador de complemento de bits, [853](#), 969  
+, 120  
+, operador de adição, 33, 35  
++, operador de incremento prefixado, [121](#)  
++, operador de pós-incremento, [121](#)  
+=, operador de atribuição de adição, [120](#), 121, 706  
sobrecarregado, 453, 454  
<!--, tag inicial de comentários XHTML, [1065](#)  
<, operador menor que, 38  
<<, operador de deslocamento para a esquerda, [853](#)  
<<, operador de inserção de fluxo, [28](#), [29](#), 33, 34  
<<=, operador de atribuição de deslocamento para a esquerda, [860](#)  
<=, operador menor que ou igual a, 38  
-=, operador de atribuição de subtração, 121  
=, operador de atribuição, 33, 162  
==, operador de igualdade, 38, 162, 454  
->\*, operador de ponteiro para membro de classe, [972](#)  
>, operador maior que, 38  
->, seleção de membro via ponteiro, 972  
-->, tag final de comentários XHTML, [1065](#)  
>=, operador maior que ou igual a, 38  
>>, operador de deslocamento para a direita, [853](#)  
>>, operador de extração de fluxo, [31](#), [33](#)  
>>=, deslocamento para a direita com operador de atribuição com extensão de sinal, [860](#)

## Números

0 (nulo), ponteiro, 812  
0 inicial, 630  
0x inicial e 0X inicial, 630  
0X, prefixo para um número hexadecimal, 626  
0x, prefixo para um número hexadecimal, 626  
2-D, array, [279](#)

## A

a, elemento âncora XHTML, [1068](#), 1072  
abort, função, [392](#), [653](#), 658, 1009, [1026](#)  
abortar um programa, 663  
abreviações em inglês, 5  
abreviando expressões de atribuição, 120  
abrir um arquivo inexistente, 673  
abrir um arquivo para a entrada, 673  
abrir um arquivo para a saída, 672, 673  
abrir um arquivo, 671, 674, 675  
abstração de dados, 412, [440](#), 458  
ação de um objeto, 170  
ação, [2](#), [94](#), 96, 99, 100, 103, 440  
Account, classe  
estudo de caso ATM, 83, 85, 86, 124, 229, 293, 294, 295, 296, 405  
exercício de hierarquia de herança, 544  
exercício, 91  
accumulate, algoritmo STL, 901, 932, [934](#), 953, 955  
acessando funções-membro de um objeto por meio de cada tipo de handle de objeto, 383  
acessar dados do chamador, 210  
acessar membro de estrutura, 848  
acessar membro private de uma classe, 64, 65  
acessar membros de dados e funções-membro não-static, 440  
acessar membros union, 1014  
acessar variável global, 215  
acesso direto, 801  
acesso indexado, 911  
acesso inválido para armazenamento, 1009  
acesso verificado, [705](#), 705  
acrescentar dados a um arquivo, 672  
action (XHTML form), atributo, [741](#), 742, [1084](#)  
acumulador, 362, 363  
Ada Lovelace, 7  
Ada, linguagem de programação, [7](#)  
adaptador (STL), [892](#), 919  
adaptador de contêiner, [893](#), 898, 919  
adaptadores de contêiner  
priority\_queue, [922](#)  
queue, [921](#)  
stack, [919](#)  
adiamento indefinido, [337](#)  
adição, 3, 34, 35, 36  
adicionar posição da memória na pilha, 839  
adicionar um inteiro a um ponteiro, 330  
adicionar uma nova conta a um arquivo, 695  
adjacent\_difference, algoritmo STL, 901  
adjacent\_find, algoritmo STL, 901, 949  
ADT (abstract data type — tipo de dados abstrato), [440](#), 441, 442  
agregação na UML, 170, 171  
agregação, [85](#), [381](#)  
aleatorização, [195](#)  
aleatorizando o programa de lançamento de dados, 196  
'além do fim', iterador, 934  
alerta ('\\a'), seqüência de escape de, 29, 1080  
alfabetar strings, 350  
<algorithm>, arquivo de cabeçalho, 191, [905](#)  
algoritmo cresça e brilhe, 94  
algoritmo de avaliação de expressão pós-fixa, 831  
algoritmo de avaliação pós-fixa, 839, 843  
algoritmo de conversão de infixo para pós-fixo, 831, 839, 843  
algoritmo de distribuição de cartas, 338  
algoritmo de uma passagem, 898  
algoritmo não-modificador de seqüência, 901

- ul style="list-style-type: none; padding-left: 0;">
- algoritmo numérico, 953
- algoritmo, xxxii, xxxiii, [94](#), 99, 100, 104, 110, 111, 799, [892](#), 899, 924
  - ação a executar, [94](#)
  - classificação do tipo *bucket sort*, 796
  - classificação por borbulhamento (*bubble sort*), 796
  - classificação por inserção, [278](#), 788
  - classificação por intercalação, [788](#)
  - classificação por seleção, [324](#), 787
  - de pesquisa binária, [783](#)
  - de pesquisa linear, 782
  - ordem em que as ações executam, [94](#)
  - pesquisa binária recursiva, 797
  - pesquisa linear recursiva, 797
  - procedimento, [94](#)
  - quicksort*, 797
- algoritmo-padrão, 899
- algoritmos de classificação, 934, 935
  - classificação do tipo *bucket sort*, 796
  - classificação por borbulhamento, 796
  - classificação por inserção, [278](#), 788
  - classificação por intercalação, [788](#)
  - classificação por seleção, [324](#), 787
  - quicksort*, 797
- algoritmos de pesquisa e classificação básica da Standard Library, 934, 935
- algoritmos de pesquisa, 934, 935
  - pesquisa binária recursiva, 797
  - pesquisa binária, [783](#)
  - pesquisa linear recursiva, 797
  - pesquisa linear, 782
- algoritmos genéricos, 899
- algoritmos matemáticos da Standard Library, 932
- algoritmos matemáticos, 932
- algoritmos modificadores de sequência, [901](#)
- algoritmos numéricos, 901
- algoritmos STL
  - accumulate*, 901, 932, [934](#), 953, 955
  - adjacent\_difference*, 901
  - adjacent\_find*, 901, 949
  - binary\_search*, 934, [936](#), 953
  - copy*, 901, [905](#)
  - copy\_backward*, 901, 938, [938](#)
  - copy\_backward*, *merge*, *unique* e *reverse*, 938
  - count*, 901, 932, [932](#)
  - count\_if*, 901, 932, [932](#)
  - equal*, 901, 926, [926](#)
  - equal*, *mismatch* e *lexicographical\_compare*, 926
  - equal\_range*, 943, [944](#), 944
  - fill*, 901, [924](#)
  - fill\_n*, 901, [924](#)
  - find*, 901, 934, [934](#)
  - find\_each*, 901
  - find\_end*, 901
  - find\_first\_of*, 901
  - find\_if*, 901, 934, [936](#)
  - for\_each*, 932, [934](#)
  - generate*, 901, [924](#)
  - generate\_n*, 901, [924](#)
  - includes*, 940, [940](#)
  - inner\_product*, 901
  - inplace\_merge*, 940, [940](#), 941
  - inplace\_merge*, *unique\_copy* e *reverse\_copy*, 940, 941
  - iter\_swap*, 901, 937, [938](#)
  - lexicographical\_compare*, 926, [926](#)
  - lower\_bound*, 943, [943](#), 944
  - make\_heap*, [946](#)
  - max*, [948](#)
  - max\_element*, 932, [934](#)
  - merge*, 938, [938](#)
  - min*, [948](#)
  - min\_element*, 932, [932](#)
  - mismatch*, 901, 926, [926](#)
  - partial\_sort*, 949
  - partial\_sort\_copy*, 949
  - partial\_sum*, 901
  - partition*, 901
  - pop\_heap*, [948](#)
  - push\_heap*, [946](#), 948
  - random\_shuffle*, 901, 932, [932](#)
  - remove*, 901, 927, [927](#), 928
  - remove\_copy*, 901, 927, [928](#), 928
  - remove\_copy\_if*, 901, 927, 928, [930](#)
  - remove\_if*, 901, 927, [928](#), 928
  - replace*, 901, 930, [930](#)
  - replace\_copy*, 901, 930, [930](#)
  - replace\_copy\_if*, 901, 930, [932](#)
  - replace\_if*, 901, 930, [930](#)
  - reverse*, 901, 938, [940](#)
  - reverse\_copy*, 901, 940, [940](#), 941
  - rotate*, 901, 949
  - rotate\_copy*, 901, 949
  - search*, 901, 949
  - search\_n*, 901, 949
  - set\_difference*, 940, [940](#)
  - set\_intersection*, 940, [940](#)
  - set\_symmetric\_difference*, 940, [940](#)
  - set\_union*, 940, [943](#)
  - sort*, 934, [936](#)
  - sort\_heap*, [946](#)
  - stable\_partition*, 901
  - stable\_sort*, 949
  - swap*, 901, 937, [937](#)
  - swap\_ranges*, 901, 937, [938](#)
  - transform*, 901, 932, [934](#)
  - unique*, 901, 938, [938](#)
  - unique\_copy*, 901, 940, [940](#), 941
  - upper\_bound*, 943, [943](#), 944
- algoritmos, formulando, 104, 109
- alias, 212, 314, 719
  - para o nome de um objeto, 395
- alinhamento à direita, [149](#), 338, 626, 627
- alinhamento à esquerda e à direita com os manipuladores de fluxo *left* e *right*, 627
- alinhamento à esquerda, [149](#), 338, 626, 627
- alinhamento de armazenamento, 848
- alinhamento, 848
- alocação dinâmica de memória, 800, 801, 1011
- alocador, 908
- alocar e desalocar dinamicamente memória, 391
- alocar memória dinamicamente, 430, [430](#), 434, 435, 659
- alocar memória, 191
- alt* (XHTML), atributo do elemento *img*, [1072](#)
- alterar o fluxo de controle, 159
- ALU, [3](#)
- alunos, programa de análise de enquete com, 262
- ambiente chamador, 674
- ambiente de desenvolvimento de programa, 9
- ambiente de programação, 183
- ambiente host, 1007
- ambiente, 183
- ambigüidade, problema de, 973, 976
- American National Standards Institute (ANSI), 2, 6, 441
- American Standard Code for Information Interchange (ASCII), [154](#), 350, 351
- amizade é concedida, não aceita, 427
- amizade não é simétrica, 427
- amizade não é transitiva, 427
- &*, caractere especial (XHTML), 1074
- análise e design de sistemas estruturados, 8
- análise e projeto orientados a objetos (*object-oriented analysis and design* – OOAD), 18
- análise, 18
- âncora de correio eletrônico, 1068
- and, palavra-chave de operador, [969](#)
- and\_eq, palavra-chave de operador, [969](#)
- aninhado dentro de um loop, [117](#)
- ANSI (American National Standards Institute), 6
- ANSI/ISO 9899: 1990, 6
- ANSI/ISO C++ Standard, xxii, 11, 956, 1060, 1062
- any, função da classe *bitset*, 950
- Apache HTTP Server, 728, [731](#)
- Apache Software Foundation, 731
- apelido, 314
- aplicativo de acesso instantâneo, [680](#)
- aplicativo de múltiplas camadas, [729](#)
- aplicativo de *n* camadas, [729](#)
- aplicativo robusto, [645](#), 649
- aplicativos cliente/servidor distribuídos, 4
- apontando um ponteiro de classe derivada para um objeto de classe básica, 554, 555
- append, função da classe *string*, [706](#)
- Apple Computer, Inc., 4
- apresentação de caractere, 192
- apresentação de um documento, [1064](#)
- Aprimorando a classe *Date*, exercício, 409
- Aprimorando a classe *Rectangle*, exercício, 409
- Aprimorando a classe *Time*, exercício, 408, 409
- área de texto (XHTML), [1086](#)
- área indefinida na memória, 848
- argc, [1005](#)
- argumento de função-membro, [60](#)
- argumento de referência, 315
- argumento implícito, 429
- argumento(s)
  - coerção de, [189](#)
  - para uma função, [60](#)
  - para uma macro, [1023](#)
  - passado para construtor de objeto-membro, 421
- argumento-padrão, [213](#), 213, 214, 215, 386
- argumentos mais à direita (finais), 213
- argumentos na ordem correta, 188
- argumentos-padrão, construtores com, 386
- argv[], [1005](#)



- 'aridade' de um operador, 452, 453
- aritmética de inteiros, 451
- aritmética de ponteiros, **321**, 330, 331, 332, 333, 904
  - dependente de máquina, 330
  - em um array de caracteres, 331
- aritmética de ponto flutuante, 451
- armazenamento livre, **431**
- armazenamento, limite de unidade de, 862
- arquivo de acesso aleatório, 669, **680**, 681, 682, 687, 688, 689
- Arquivo de cabeçalho da classe `BasePlusCommissionEmployee`, 575
- arquivo de cabeçalho de fluxo de entrada/saída `<iostream>`, **28**
- arquivo de cabeçalho, **70**, 78, 190, 377, 378, 540, 582, 753, 1007, 1022
  - `<algorithm>`, 905
  - `<deque>`, **911**
  - `<exception>`, 191, **646**, 653, 661
  - `<fstream>`, 191, 671
  - `<functional>`, 191, **953**, 953
  - `<iomanip>`, 116, 191, 614, 621, 1022
  - `<iostream>`, 27, **28**, 155, 191, 614, 671, 966, 1022
  - `<iterator>`, 191, 938, 940
  - `<limits>`, 192
  - `<list>`, 191, 895, **908**
  - `<locale>`, 192
  - localização, 73
  - `<map>`, **915**, 918
  - `<memory>`, 191, **659**
  - `<new>`, 656
  - nome de, entre aspas (" "), 73
  - nome de, entre colchetes angulares (< >), 73
  - `<numeric>`, **901**, 934
  - `<queue>`, 191, 895, **922**, 923
  - `<set>`, 191, 895, 913, 915
  - `<sstream>`, 191, **719**
  - `<stack>`, 191, 895, **920**
  - `<stdexcept>`, 191, **646**, 661
  - `<string>`, **60**, 72, 191, 484, 704
  - `<typeinfo>`, 191, **585**
  - `<vector>`, 288
- arquivo de código-fonte, **70**
- arquivo de código-objeto pré-compilado, 444
- arquivo de contas a receber, 670
- arquivo de folha de pagamento, 670
- Arquivo de implementação da classe `BasePlusCommissionEmployee`, 552, 575
- arquivo de implementação, 444
- arquivo de *n* bytes, 671
- arquivo de objeto, 582
- arquivo de texto, 689
- arquivo de transações, 699
- arquivo do lado do servidor, **755**, 761
- arquivo em disco, 843
- arquivo sequencial, 669, **670**, 671, 672, 674, 675, 680
- arquivo, **669**, 670, 671, 675
  - como uma sequência de bytes, 671
- arquivo, escopo de, **202**, 202, **382**, 436, 606, 1014
- arquivo-fonte, 1007
- arquivos de cabeçalho 'no estilo antigo', 190
- Arquivos temporários da Internet, diretório, 753
- array associativo, **918**
- array automático, 253
- array baseado em ponteiro no estilo C, 893, 924
- array bidimensional, **279**, 279, 280, 282, 799
- array bruto, 441
- array de caracteres, 263, 264, 333, 346, 455, 717, 718
  - como uma string, 264, 265
- array dinâmico, **1011**
- 'array inteligente', 263
- array local automático, 264
- array multidimensional, **279**, 280, 281
- array, **251**, 252, 322, 441, 442, 801
  - atribuição de, 441
  - comparação de, 441
  - de ponteiros para funções, 344, 345
  - de string, 335
  - entrada/saída de, 441
  - limites de, 263
  - lista inicializadora, 253, 254, 255
  - nome como um ponteiro constante para o começo do array, 323, 332, 333
  - nome, 332, 333
  - notação para referenciar elementos, 333
  - operador de subscrito de, (`[]`), 465
  - passado por referência, 269
  - subscrito de, 323, 333
  - tamanho, 267
  - uso de, com funções, 268
  - verificação de limite de, 263
- Array, classe, 458
  - definição da, com operadores sobrecarregados, 459, 489
  - definições de função-membro e função `friend` da, 459
- arrays que sabem seus tamanhos, 441
- arredondar um número de ponto flutuante para propósitos de exibição, 116
- arredondar, **116**, 184
- artefato na UML, **1057**
- árvore binária, **799**, **820**, 821, 945
  - classificação de, **827**, 845
  - com duplicatas, 833
  - exclusão de, 834
  - percurso na ordem de nível, 828, 834
  - pesquisa de, 834
- árvore de pesquisa binária, **821**, 825, 827, 833, 834
- árvore equilibrada, 828
- árvore, 442, 800, 820, 827, 828
- ASCII (American Standard Code for Information Interchange), **154**, 346, 350, 351, 616
  - conjunto de caracteres, **54**
  - equivalente, decimal de um caractere, 845
- `asctime`, função, 734, **735**
- aspas duplas, 28, 29
- aspas francesas (« e ») na UML, 70
- aspas simples ('), 29, 346
- aspas vazias, 682
- aspas, 28, 29
- assembler, **5**
- `assert`, **1026**
- `<assert.h>`, arquivo de cabeçalho, 192
- `assign`, função-membro da classe `string`, **705**
- `assign`, função-membro de `list`, **911**
- assinatura, **189**, 216, 217, 479
- assinaturas das funções operadoras sobrecarregadas de incremento prefixado e pós-fixado, 479
- associação (na UML), **18**, **84**, 85, 401
  - nome da, 84
- associação em um map, 918
- associando um fluxo de saída a um fluxo de entrada, 636
- associatividade
  - da direita para a esquerda, 40, 123, 155
  - da esquerda para a direita, 40, 41, 123, 161, 164
  - dos operadores, **36**, 40, 41, 164
  - gráfico de, 41
  - não alterada pela sobrecarga, 452
- assumir o padrão de acesso `public`, 847
- assumir padrão de decimal, 630
- asterisco (\*) para multiplicação, **35**
- AT&T, 8
- `at`, função-membro
  - da classe `string`, **705**, 706, 719
  - da classe `vector`, **292**, 907, 908, 950
- `atexit`, função, **1007**, 1007, 1008
- ativação em um diagrama de seqüências na UML, **295**
- atividade (UML), **46**, **169**, 171
- atividade de uma parte de um sistema de software, **96**
- atividades paralelas, 8
- ativo de software, 18
- ATM (*automated teller machine*), estudo de caso, 41
- ATM, classe (estudo de caso ATM), 83, 84, 85, 124, 125, 126, 169, 229, 292, 293, 294, 295, 400
- ATM, sistema, 44, 45, 82, 83, 123, 124, 169, 229, 399, 400
- `atof`, função, 869, **869**
- `atoi`, função, **745**, 754, 841, 843, 869, **869**
- `atol`, função, 869, **870**
- ator em casos de uso (UML), **45**
- atribuição de bits, operadores de, **859**, 950
  - palavras-chave, 969
- atribuição de membro a membro, **397**, 452
- atribuição de objetos de classe, 397
- atribuição de ponteiro, 331, 332
- atribuição-padrão de membro a membro, 397, 452, 895
- atribuições em cascata, 477
- atribuindo endereços de objetos das classes básica e derivada aos ponteiros de classe básica e derivada, 553
- atribuir strings de caracteres a objetos `String`, 469
- atribuir um iterador para outro, 900
- atribuir um valor (*set*), 66
- atribuir uma estrutura a uma estrutura do mesmo tipo, 848

atribuir uma union a outra union do mesmo tipo, 1014  
 atributo, 17, **62**, 63, 401  
   compartimento em um diagrama de classes, 125  
   de um elemento (XHTML), 1065, 1066  
   declaração de, na UML, 125  
   na UML, **17**, 17, 59, 83, 86, 123, 124, **125**, 126, 169, 171, 172, 587  
   nome do, na UML, 125  
 atributos de uma variável, 200  
 atualizar registros no local, 680  
 atualizar um registro, 700  
 autenticação, informações de, 728  
 auto, especificador de classe de armazenamento, **200**, 200  
 auto\_ptr, classe, **659**, 659, 661  
 auto\_ptr, template de classe, 659  
 auto-atribuição, 430, **467**, 467, 477  
 autodocumentado, programa, **32**  
 automaticamente destruída, variável, 202  
 automóvel, 139  
 Autos, janela, exibe a variável local withdrawalAmount atualizada, 1109  
 Autos, janela, exibe a variável local withdrawalAmount, 1109  
 Autos, janela, exibe o estado do objeto account1 depois da inicialização, 1108  
 Autos, janela, exibe o estado do objeto account1, 1108  
 Autos, janela, **1108**  
   exibe estados de objetos, 1108  
 avaliação da esquerda para a direita, 36, 37  
 avaliação de curto-circuito, **161**  
 avaliador de pós-fixado, 842, 845  
 avaliar expressões, 814, 831  
 avaliar uma expressão pós-fixada, 832  
 avanço de formulário ('\\f'), 864, 866  
 aviso sonoro do sistema, 29  
 aviso, 29  
**B**  
 B, linguagem de programação, 6  
 Babbage, Charles, 7  
 back, função-membro  
   de contêineres de sequência, **902**, 905  
   de queue, **921**  
 back\_inserter, template de função, **938**, 940  
 bad (fluxos), função-membro, 634  
 bad\_alloc, exceção, **656**, 657, 658, 661, 800, 908  
 bad\_cast, exceção, **661**  
 bad\_exception, exceção, **661**  
 bad\_typeid, exceção, **661**  
 badbit de fluxo, **617**, 634, 673, 674  
 BalanceInquiry, classe (estudo de caso ATM), 83, 85, 124, 126, 170, 229, 293, 294, 295, 401, 585, 586, 587  
 banco de dados protegido por senha, 761  
 banco de dados, **671**  
 BankDatabase, classe (estudo de caso ATM), 83, 85, 124, 229, 230, 293, 294, 295, 296, 400, 401  
 baralho, 335, 336, 337  
 barra de asteriscos, 258, 259

barra de título, **1066**  
 barra indicadora marginal, **1100**  
 barra invertida (\\), 29, 1024  
 barra normal (/), **1072**  
 barras invertidas (\\), sequência de escape de, 29  
 base 10, sistema de numeração de, 184, 626  
 base 16, sistema de numeração de, 626  
 base 2, 853  
 base 8, sistema de numeração de, 626  
 base de fluxo, 621  
 base e, 184  
 base especificada para um fluxo, 630  
 BASIC (Beginner's All-Purpose Symbolic Instruction Code), 8, 799, 835, 845  
 basic\_fstream, template, **615**, 671  
 basic\_ifstream, template, **615**, 671  
 basic\_ios, template de classe, 614, 978  
 basic\_iostream, template de classe, 614, 615, 671, 978  
 basic\_istream, template de classe, 614, 671, 978  
 basic\_istream, template de classe, 719  
 basic\_ofstream, template de classe, **615**, 671  
 basic\_ostream, template de classe, 614, 615, 671, 978  
 basic\_ostringstream, template de classe, 719  
 basic\_string, template de classe, **704**  
 BCPL, linguagem de programação, 6  
 begin, função-membro da classe string, **718**  
 begin, função-membro de contêineres de primeira classe, **896**  
 begin, função-membro de contêineres, 894  
 Beginner's All-Purpose Symbolic Instruction Code (BASIC), 8  
 Bell Laboratories, 6  
 bibliografia, 11  
 biblioteca de classe, 8, 149, 382, 399, 540  
 biblioteca de tratamento de sinal, **1009**  
 biblioteca de utilitários gerais <cstdlib>, **868**, 869, 1007, 1011, 1026  
 bibliotecas de classes padrão, 441  
 bibliotecas de fluxo clássicas, **613**  
 bibliotecas de fluxo padrão, **613**  
 bibliotecas privadas, 10  
 binary digit, **669**  
 binary\_function, classe, **953**  
 binary\_search, algoritmo STL, 934, **936**, 953  
 bit de transporte, 997  
 bit, **669**, 847  
 bitand, palavra-chave de operador, **969**  
 bitor, palavra-chave de operador, **969**  
 bits 'mascarados', 854, 855  
 bits de erro, 619  
 bits de estado, **617**  
 bits de status, 635  
 'bits e bytes', nível dos, 852  
 bits, operadores de, 852, 853, 859  
   palavras-chave, 969  
 bitset  
   flip, **951**  
   reset, **951**  
 bitset, classe, 893, 950, **950**, 951  
 <bitset>, arquivo de cabeçalho, 191, 895  
 bloco catch correspondente, 649

bloco de dados, 874  
 bloco de memória, 874, 911, 1012  
 Bloco de Notas, editor de textos, 1064  
 bloco está ativo, 200  
 bloco externo, 202  
 bloco interno, 202  
 bloco try expira, 649  
 bloco, **40**, 81, **102**, 103, 114, 115, 200, 202  
 bloco, escopo de, **202**, 202, 382  
   variável de, 382  
 bloco, saindo de um, 200  
 blocos aninhados, 202  
 blocos de construção, abordagem de, 6  
 blocos de construção, aparência dos, 165  
 'blocos de construção', 18  
 Boas práticas de programação, **6**, 11  
 body (XHTML), elemento, **1065**, 1065, 1066  
 Böhm, C., 95, 96, 167  
 'bombing', **110**  
 Booch, Grady, 18, 19  
 bool true, valor, 98, 99  
 bool, tipo de dados, **98**  
 boolalpha, manipulador de fluxo, **163**, 632  
 Boolean, atributo, na UML, **124**  
 border, atributo do elemento table, **1077**  
 Borland C++ Builder, 956  
 Borland C++, 10, 217, 1007  
 botão de opção (XHTML), **1086**  
 br (XHTML), elemento, **1074**  
 branch negative, 839  
 branch zero, instrução de, 839, 841  
 break, comando de depurador, **1114**  
 break, instrução, 155, 156, 159, 179, 180, 1012  
   saindo de uma instrução for, 159  
 bucket sort, classificação do tipo, 309, **309**, 796  
 buffer de saída, 636  
 buffer é esvaziado, 615  
 buffer é preenchido, 636, 615  
 bug, **1098**  
 button (type), valor do atributo, 742  
 byte, **669**, **852**

## C

C ANSI, 11  
 C#, linguagem de programação, **8**, **8**  
 C, alocação dinâmica de memória no estilo, 1011  
 C, array baseado em ponteiro no estilo, 893, 924  
 C, código legado, 412, 1002, 1011, 1022, 1023, 1026  
 .C, extensão, 10  
 C, linguagem de programação, 5, 1064  
 C, strings char \* no estilo, 717  
 c\_str, função-membro da classe string, **718**, 737  
 C++ Standard Library, 6, 183  
   arquivo de cabeçalho <string>, **60**  
   arquivos de cabeçalho, 190  
   classe string, **60**  
   localização de arquivo de cabeçalho, 73  
   template de classe vector, 288  
 C++, 4, 5, 6, 11  
   ambiente de desenvolvimento, 9  
   ambiente de programação, 183  
   ambiente, 9

- arquivo executável, 735
- compilador, 10
- linguagem de programação, 8
- linguagem, 11
- pré-processador, 10, 28
- recursos de carreira, 1060
- recursos na Web, 11, 21, 1059
- cabeça (parte da frente) de uma fila, **799**, 817
- cabeçalho da tabela (XHTML), **1081**
- cabeçalho de argumentos de variáveis `<csdarg>`, **1003**
- cabeçalho, 1066
- cache, **729**
- caixa de seleção (XHTML), **1086**, 1086
- caixa de senha, 1086
- caixa de texto, 1084, **1086**, 1086
- caixa eletrônico (*automated teller machine* – ATM), 41, 45, 680
  - interface com o usuário, 42
- caixa, 53
- calculando a soma dos elementos de um array, 258
- calcular o valor de  $\pi$ , 179
- calcular o valor mínimo em um array, exercício de recursão, 228
- calcular os rendimentos de um vendedor, 135
- cálculo matemático, 183
- cálculos aritméticos, 35
- cálculos matemáticos, 7
- cálculos monetários, 149
- cálculos, 3, 35, 96
- `calloc`, 1011, 1012
- camada de cliente, **730**
- camada de dados, **730**
- camada de informações, **730**
- camada inferior, **730**
- camada intermediária, **730**
- camada superior, **730**
- camada, **729**
- caminho de um recurso, 732
- campo alinhado, 628
- campo de bit com uma largura de zero, **862**
- campo de bit não identificado com uma largura de zero, **862**
- campo de bit não identificado, **862**
- campo de bit, 852, **859**, 862
  - manipulação, 862
  - membro de estrutura, 861
  - para economizar espaço, 863
- campo de tipo, 696
- campo, **669**
- campos maiores do que valores impressos, 629
- capacidade de uma string, **710**
- capacidades de E/S de baixo nível, 613
- `capacity`, função-membro de uma string, **710**
- `capacity`, função-membro de vector, **903**, 904
- capital, 180
- `caption` (XHTML), elemento, **1077**
- captura (catch) de todas as exceções, 662
- capturar erros relacionados, 654
- capturar um objeto de classe básica, 661
- caractere constante, **346**
- caractere de controle, 866
- caractere de escape, **29**
- caractere de espaço em branco, 28, 40, 98, 156, 264, **616**, 618, 864, 866, 1022
  - espaço em branco, 98
  - nova linha, 98
  - tabulação, 98
- caractere de impressão diferente de um espaço, 864
- caractere de impressão diferente de um espaço, dígito ou letra, 866, 867
- caractere de impressão incluindo espaço, 864
- caractere de nova linha, 98, 156
- caractere de preenchimento, **378**, **622**, 626, 628
- caractere de terminação nulo, `'\0'`, de uma string, 426
- caractere de terminação nulo, 263, 264, 346, 347, 348, 352, 353, 717, 718
- caractere especial, 32, **346**, 1074
  - XHTML, **1074**
- caractere nulo (`'\0'`), **263**, 263, 264, 321, 334, 336, 346, 347, 348, 352, 353, 620, 800, 832
- caractere, **669**, 847
- caracteres de preenchimento, **622**, 626, 628, 629
- caracteres representados como códigos numéricos, 351
- carregamento, 9, 10
- carrinho de compras (*shopping cart*), aplicativo de, 752, 761
- cartas, jogos de, 336
- cascata de operadores `+=`, 477
- case, rótulo, 155, 156
- CashDispenser, classe (estudo de caso ATM), 83, 84, 85, 124, 125, 229, 296
- caso básico em recursão, **221**, 222, 224, 226
- casos de uso na UML, **44**, 45
- casos de uso na UML, diagrama de, **44**, 45
- casos de uso, modelagem de, **44**
- `<cassert>`, arquivo de cabeçalho, 192, 1026
- cassino, 196
- catálogo, 399
- catch
  - handler, **649**, 649, 650, 654
  - palavra-chave, **649**
- `catch(...)`, 662
- cauda de uma fila, **799**, 817
- `<cctype>`, arquivo de cabeçalho, 191, 321, 863
- `ceil`, função, 184
- célula de cabeçalho (tabela XHTML), 1081, **1081**
- células de dados, **1081**
- `cerr` (erro-padrão não armazenado em buffer), 10, 614, 671
- `<cfloat>`, arquivo de cabeçalho, 192
- CGI (Common Gateway Interface), **731**
- `.cgi`, extensão de arquivo, 734, 735
- `cgi-bin`, diretório, 734, 735, 761
- chamada assíncrona, **294**
- chamada de função aninhada, 663
- chamada de função recursiva, 814
- chamada de função, 60, 183, 188
  - operador de, `()`, 479, 581
  - overhead de, 208, 477
  - pilha de, **204**, 322
- chamada de função-membro, **57**, 451
- chamada recursiva, **221**, 224
- chamada síncrona, **294**
- chamadas de função-membro concisas, 380
- chamadas de função-membro em cascata, **430**, 430, 432, 434
- chamadas de função-membro para objetos `const`, 413
- chamar funções por referência, 315
- `char **`, 870
- `char`, tipo de dados, 32, **154**, 190, 718, 852
- chave de classificação, **781**
- chave de pesquisa, **276**, **781**, **913**
- chave de registro, **670**, 700
- chave direita `()`, **28**, 29, 114
- chave esquerda `()`, **28**, 31
- chave, 913
- chave/valor, pares de, **893**, 915, 918
- chaves `{}`, 29, 40, 81, 102, 114, 155
  - em uma instrução `do...while`, 150
- chaves duplicadas, 913, 915
- chaves únicas, 913, 915, 918
- checkbox (type), valor do atributo, 742
- checked, atributo, 1086
- chegada de mensagem de rede, 650
- chinês, 441
- ciclo de execução de instrução, 365
- ciclo de vida do software, **44**
- `cin` (fluxo de entrada padrão), 10, 33, 614, 671, 674
  - função `clear`, 635, 636
  - função `eof`, 617, 634
  - função `get`, 154, 155, 617, 618
  - função `getline`, **347**
  - função `tie`, 636
- círculo pequeno, **96**
- círculo sólido (para representar um estado inicial em um diagrama UML) na UML, **169**, 170
- círculo sólido dentro de um círculo vazio (para representar o final de um diagrama de atividades UML), 170
- círculo sólido marrom de ponto de interrupção, 1100
- círculo sólido, **96**
- clareza, 2, 11, 33, 261
- `class`, palavra-chave, 219, 597
- classe abstrata, **563**, 563, 564, 580
- classe auto-referencial, **799**, 800
- Classe `BasePlusCommissionEmployee` que herda da classe `CommissionEmployee` mas não pode acessar diretamente os dados `private` da classe, 529
- Classe `BasePlusCommissionEmployee` que herda dados `protected` de `CommissionEmployee`, 523
- Classe `BasePlusCommissionEmployee` representa um empregado que recebe um salário-base além de uma comissão, 511
- classe básica abstrata, 547, **563**, 564, 588
- classe básica direta, **502**
- classe básica indireta, **502**, 503
- classe básica, **502**, 502, 503, 504, 586
  - acessibilidade do membro de, em uma classe derivada, 540
  - construtor de, 533
  - `exception`, 661



- função-membro de, redefinida para uma classe derivada, 530
- handler catch, 661
- membro private da, 504, 505
- ponteiro de, (ou tipo por referência), 801
- ponteiro de, para um objeto de classe derivada, 585
- sintaxe inicializadora da, **516**
- subobjeto de, 978
- classe concreta, **563**, 565, 566
- classe contêiner, 384, 426, **442**, 465, 596, 605, 893
- classe de armazenamento automática, **200**, 200, 251, 264
- classe de armazenamento estática, **200**, 200, 202
- classe de armazenamento, **200**, 201, 1005
- classe de fluxo, 671
- classe de pilha, 596
- classe derivada, **502**, 502, 503, 504, 539, 540, 586
  - catch, 661
  - destrutor de, 585
  - indireta, 574
- classe genérica, 599
- classe mais derivada, **980**
- classe proxy, 384, **442**, 442, 444, 445
- Classe Time com funções-membro const, 414
- Classe Time contendo um construtor com argumentos-padrão, 387
- classe, 6, 17, 125, 126, 229, 231, 399, 400, 1022
  - atributo, **62**, 63
  - construtor, 67, 68
  - construtor-padrão, **68**, 69, 70
  - convenção para atribuição de nomes, 58
  - definição da, **58**
  - definindo funções-membro em um arquivo de código-fonte separado, 76
  - definir um construtor, 69
  - definir uma, **58**
  - definir uma, com uma função-membro, **57**
  - desenvolvimento, 458
  - função-membro, 56, 57
  - instância de, 64
  - interface definida com protótipos de função, **74**
  - interface, **74**
  - membro de dados, 57, **62**
  - nome, 401
  - objeto de, 64
  - programador de código-cliente, 78
  - programador de implementação de, 77, 78
  - serviços public, **74**
  - serviços, 66
- classe, escopo de, **202**, **379**, 379, 382
- classes de coleção, **442**
- classes de exceções derivadas de uma classe básica comum, 654
- classes de exceções padrão, 662
- classes matemáticas, 451
- classes proprietárias (patenteadas), 540
- classes, 484
  - Array, 458, 459
  - auto\_ptr, **659**, 659
  - binary\_function, **953**
  - bitset e o crivo de Eratóstenes, 951
  - Complex, 494
  - deque, **892**
  - exception, **646**
  - HugeInt, 494, 496
  - invalid\_argument, **662**
  - list, **892**
  - Node, 799, 800
  - Polynomial, 500
  - RationalNumber, 500
  - string, **60**, 468, 469
  - vector, 288, 484
- classificação de strings, 192
- classificação por borbulhamento, **303**, 303, 796
  - aprimorada, 796
- classificação por inserção, algoritmo de, **278**, 788
- classificação por intercalação, algoritmo de, **788**, 788, 791
- classificação por intercalação, exemplo de recursão, 228
- classificação por seleção com passagem por referência, 326
- classificação por seleção, algoritmo de, **309**, **324**, 324, 787
- classificação por seleção, exercício de recursão, 227, 309
- classificação, 442, 672, 799, 934, 935
- classificar arrays, **276**
- classificar dados, **781**, 787
- clear, função de ios\_base, **635**
- clear, função-membro de contêineres de primeira classe, 908
- clear, função-membro de contêineres, 894
- cliente de um objeto, **57**, 66
- cliente de uma classe, **17**, 228, 294
- cliente de uma fila, 442
- cliente, 444, 753, 757
- cliente/servidor, computação, **4**
- clients.txt, 757
- <limits>, arquivo de cabeçalho, 192
- clog (erro-padrão armazenado em buffer), 614, 671
- close, função de ofstream, **674**
- <cmath>, arquivo de cabeçalho, 191
- cn (domínio de nível mais alto), 731
- COBOL (Common Business Oriented Language), **7**, 1064
- código de caractere, **351**
- código de cliente, 548
- código de função não modificável, 382
- código de linguagem de máquina, 149, 814
- código de operação, 362, 838
- código de processamento de erro, 646
- código legado C, 1023
- código legado, 319, 1002, 1011, 1026
- código não-otimizado, 843
- código otimizado, 844
- código portátil, 6
- código reentrante, **382**
- código-fonte aberto, software de, **731**
- código-fonte da classe, 382
- código-fonte, 10, 382, 540
- código-objeto da classe, 382
- código-objeto, **5**, 10, 77, 78, 382
- coeficiente, 500
- coerção const, 964
- coerção dinâmica, **547**
- coerção, 332
- col (XHTML), elemento, **1081**
- colaboração na UML, **292**, 293, 294
- colchetes ([]), 252
- colchetes angulares (< e >),
  - em templates, 219, **597**
  - para nomes de arquivos de cabeçalho, 1022
- coleta de requisitos, **44**
- colgroup (XHTML), elemento, **1081**
- cols (XHTML), atributo do elemento textarea, **1086**
- colspan (XHTML), atributo do elemento th, **1081**
- coluna, **279**
- com (domínio de nível mais alto), 731
- combinando a classe Time e a classe Date, exercício, 409
- comentário de uma única linha, **27**
- comentário, **27**, 28, 32
- CommissionEmployee, arquivo de cabeçalho da classe, 572
- CommissionEmployee, arquivo de implementação da classe, 573
- CommissionEmployee, classe, com dados protected, 520
- CommissionEmployee, classe, que representa um empregado que ganha uma porcentagem das vendas brutas, 507
- CommissionEmployee, classe, utiliza funções-membro para manipular seus dados private, 527
- CommissionEmployee, programa de teste da classe, 509
- Common Gateway Interface (CGI), **731**, 1084
- comparação de ponteiros, 332
- comparação lexicográfica, **709**
- comparando
  - blocos de memória, 874
  - iteradores, 900
  - strings, 347, 350, 707
  - unions, 1014
- compare, função-membro da classe string, **709**
- compartilhamento de tempo, **4**, 8
- compartimento de operação em um diagrama de classes, 229
- compilação condicional, **1022**, **1024**
- compilação, 9
- compilador de otimização, 149, 201, 412
- compilador gera instruções de SML, 839
- compilador, **5**, 29, 115, 814, 844, 845
- compilar, 799, 828, 1007
  - programas de arquivo de múltiplas fontes, 78, 1005
- comp1, palavra-chave de operador, **969**
- complemento de bits (~), operador de, **853**, 853, 855, 857, 859, 969
- complemento de dois, 997, 998
- complemento de E/S de disco, 650
- complemento de um, 857, 997
  - operador, (~), **853**
- Complex, classe, 408, 494

- definições de função-membro, 495
- exercício, 408
- complexidade exponencial, **225**
- componente de software reutilizável, 6
- componente na UML, **18**, **1057**
- componente, 399
- componentes reutilizáveis, 8, 399
- componentes, 182
- comportamento do sistema, **45**, 169, 170, 172, 294
- comportamento, 17, 228, 441
- comportamentos na UML, **17**
- composição, **84**, 85, 87, **381**, **421**, 502, 504, 815
- comprimento de uma string, 263, 346, 347, 710, 712
- comprimento de uma substring, 478
- computação baseada na força bruta, **179**
- computação conversacional, **33**
- computação de missão crítica, 649
- computação de negócios críticos, 649
- computação distribuída, **4**
- computação interativa, **33**
- computação, 3
- computador cliente, **4**, 752
- computador pessoal, **3**, **4**
- computador, **3**
- Computadores na educação, exercício, 244, 245
- comutativo, 454
- concatenar dois objetos de lista vinculada, 831
- concatenar strings, 348, 477
- concatenar, 706
- conceito, 441
- condição de continuação do loop falha, 226
- condição de continuação do loop, **97**, **141**, 141, 143, 145, 150, 151, 179, 180
- condição de guarda na UML, 98, 99, 100, 103, 104, 146, 151, 157, 170
- condição de terminação, 221, 263
- condição excepcional, 156
- condição falsa, 38, 226
- condição simples, **161**, 161
- condição verdadeira, 37
- condição, **37**, 98, 100, 150, 151, 160, 161
- condições complexas, 161
- conexão de rede, 613
- Conexões de Rede e Dialup, explorer, 731
- configuração aderente, **149**, 378
- configuração de precisão, 621
- configurações do formato original, 633
- configurar (*set*), 66
- confundindo operadores de igualdade (==) com operadores de atribuição (=), 38, 164
- conjunto de caracteres, **54**, 158, 351, **669**
- conjunto de chamadas recursivas à função fibonacci, 225
- const char \*, 321
- const com parâmetros de função, 319
- const de operator[], versão, 468
- const, 268, 269, 270, 412, 413, 457, 1023
- const, 970, 971
- const, função-membro, 412
- const, função-membro, em um objeto const, 413
- const, função-membro, em um objeto não-const, 413
- const, objeto, 412, 413
- const, objetos, e funções-membro const, 416
- const, palavra-chave, 209
- const, ponteiro, 288, 458
- const, qualificador de tipo, aplicado a um parâmetro de array, 270
- const, qualificador, **255**, 319, 964
- const, qualificador, antes do especificador de tipo na declaração de parâmetro, 211
- const, variáveis, devem ser inicializadas, 257
- const\_cast
  - coerção const, 964
- const\_cast, operador, **964**, 965, 970, 971
- const\_iterator, **718**, 718, 719, 894, 895, 896, 899, 904, 905, 915, 918
- const\_reference, 895
- const\_reverse\_iterator, **718**, 894, 895, 899, 905
- constante de ponto flutuante não sufixada, 1009
- constante de ponto flutuante, **115**
- constante enumerada, **197**, 1024
- constante identificada, **255**
- constante simbólica NDEBUG, 1026
- constante simbólica PI, 1023
- constante string, **346**
- constante, 838
  - de ponto flutuante, **115**
- constantes simbólicas predefinidas, **1026**, 1026
- constantes simbólicas, **1022**, 1023, 1024, 1026
- construído de dentro para fora, 422
- construindo seu próprio compilador, 799, 835
- construindo um compilador, 836
- construtor de cópia, **398**, **465**, 466, 467, 894, 895
  - em passagem de parâmetros de chamada por valor, 477
- construtor, **67**, 68
  - argumentos-padrão, 390
  - atribuição de nomes, 69
  - chamado recursivamente, 466
  - de conversão, **468**, 469, 477, 488
  - de um único argumento, 468, **469**, 487, 488
  - definindo um, 69
  - em um diagrama de classe UML, 70
  - em uma union, 1014
  - explicit, **488**, 489
  - lista de parâmetros, 69
  - não pode especificar um tipo de retorno, 68
  - não pode ser virtual, 585
  - padrão, 69, 70
  - protótipo de função, 74
- construtores de objeto global, 392
- construtores e destrutores chamados automaticamente, 392
- construtor-padrão do objeto-membro, 425, 426
- construtor-padrão, **68**, 69, 70, 387, 425, 426, 464, 466, 480, 602, 894
  - criado pelo compilador, 70
  - definido pelo programador, 70
- consumo de combustível de automóveis, 133
- contador de instrução, 839, 841
- contador de loop, 141
- contador, **104**, 117, 135, 201
- contagem baseada em zero, **144**
- contagem do loop, 142
- conta-poupança, 147
- contêiner associativo, **893**, 895, 898, 913, 915
- contêiner de primeira classe, **892**, 894, 895, 896, 897, 905, 908
  - função clear, 908
  - função erase, 907
  - função-membro begin, **896**
  - função-membro end, **896**
- contêiner de sequência, **893**, 898, 901, 907, 908, 911
  - função back, **902**, 905
  - função empty, **907**
  - função front, **902**, 905
  - função insert, **907**
- contêiner sequencial, **893**
- contêiner subjacente, 919, 920
- contêiner, xxxii, xxxiii, 191, 799, **892**, 892, 893, 924
- contêineres
  - função begin, 894
  - função clear, 894
  - função empty, 894
  - função end, 894
  - função erase, 894, 907
  - função max\_size, 894
  - função rbegin, 894
  - função rend, 894
  - função size, 894
  - função swap, 894
- CONTENT\_LENGTH, variável de ambiente, **745**
- Content-Type, cabeçalho, **733**, 735, 745
- conteúdo dinâmico, 7
- conteúdo Web dinâmico *versus* conteúdo Web estático, 731
- conteúdo Web dinâmico, **731**, 731, 733
- conteúdo Web estático, **731**
- continue (depurador GNU C++), comando, **1115**
- Continue (depurador Visual C++ .NET), comando, 1101, **1101**
- continue, instrução, 159, 179, 180
  - que termina uma única iteração de uma instrução for, 160
- controlando a impressão de zeros finais e pontos de fração decimal para doubles, 626, 627
- controle do programa, **94**
- convergir para o caso básico (recursão), 226
- conversão definida por compilador implícita entre tipos fundamentais, 477
- conversão entre um tipo fundamental e uma classe, 477
- conversão explícita, **115**
- conversão implícita inadequada, 487
- conversão implícita, **115**, 469, 487, 488
  - via construtores de conversão, 488
- conversão métrica, programa de, 374
- conversões entre tipos fundamentais, 468
  - por coerção, 468, 469
- conversões implícitas definidas pelo usuário, 469
- conversor de infixo para pós-fixado, 845
- convertendo strings em strings no estilo C e arrays de caracteres, 717

Convertendo uma string em letras maiúsculas, 320  
 converter de um tipo de dados mais alto em um tipo mais baixo, 190  
 converter entre tipos definidos pelo usuário e tipos fundamentais, 468  
 converter entre tipos, 468  
 converter letras minúsculas, 191  
 converter um número binário em decimal, 996  
 converter um número hexadecimal em decimal, 996  
 converter um número octal em decimal, 996  
 converter, 10  
 cookie CART, 767  
 cookie, xxxi, 752, 753, 761  
 cópia de membro a membro padrão, 466, 895  
 cópia de membro a membro, 466  
 cópia de strings utilizando a notação de array, 335  
 cópia de strings utilizando a notação de ponteiro, 335  
 cópia do argumento, 319  
 cópia não incrementada de um objeto, 484  
 copiando os exemplos do livro a partir do CD-ROM, xxxix  
 copiando strings, 333, 335, 347  
 'copiar e colar', abordagem, 515  
 copy, algoritmo STL, 901, 905  
 copy, função-membro da classe string, 718  
 &copy; (XHTML), caractere especial, 1074  
 copy\_backward, algoritmo STL, 901, 938, 938  
 coroa, 192  
 corpo da tabela (tabela XHTML), 1081  
 corpo da tabela (XHTML), 1081  
 corpo de múltiplas instruções, 40  
 corpo  
   de um loop, 103, 141, 142, 145, 179  
   de uma definição de classe, 58  
   de uma função, 28, 29, 59  
 correção, 10  
 correio eletrônico (e-mail), 1068, 1070  
 corretor ortográfico, 889  
 cos, função, 184  
 co-seno trigonométrico, 184  
 co-seno, 184  
 count, algoritmo STL, 901, 932, 932  
 count, função de contêiner associativo, 913, 915  
 count\_if, algoritmo STL, 901, 932, 932  
 cout (<<) (fluxo de saída padrão), 10, 28, 30, 31, 33, 614, 671  
 cout, função-membro put, 616  
 cout, função-membro write, 620  
 \_\_cplusplus, constante simbólica predefinida, 1026  
 .cpp, extensão, 10  
 CPU, 3, 10  
 'crashing', 110  
 CreateAndDestroy, classe  
   definição, 392  
   definições de função-membro, 393  
 Criando e percorrendo uma árvore binária, 825  
 Criando um arquivo de acesso aleatório com 100 registros em branco sequencialmente, 685  
 Criando um arquivo sequencial, 672

criar novos tipos de dados, 34, 440, 441  
 criar um arquivo de acesso aleatório, 681  
 criar uma associação, 1147  
 criptografia de chave simétrica, 724, 724  
 criptografia, 138, 724, 725  
 criptograma, 725  
 Crivo de Eratóstenes, 309, 951, 961  
 <setjmp>, arquivo de cabeçalho, 663  
 <signal>, arquivo de cabeçalho, 1009  
 <signal>, arquivo de cabeçalho, 1009  
 <stdio>, arquivo de cabeçalho, 192  
 <stdlib>, arquivo de cabeçalho, 191, 192, 658, 737, 745, 868, 869, 1007, 1011  
 <string>, arquivo de cabeçalho, 191, 347, 478  
 <time>, arquivo de cabeçalho, 191, 195, 734  
 Ctrl, tecla, 155  
 <ctrl>-c, 1009  
 <ctrl>-d, 155, 674  
 <ctrl>-z, 155, 623, 674  
 <type.h>, arquivo de cabeçalho, 191  
 cursor de tela, 29  
 cursor, 29, 29  
 curto-circuito, avaliação de, 161  
 .cxx, extensão, 10

## D

dado de seis faces, 192  
 dados brutos, 680  
 Dados de classe básica protected podem ser acessados a partir da classe derivada, 524  
 dados de tabela (tabela XHTML), 1081  
 dados, 3  
 data em que o arquivo-fonte é compilado, 1026  
 data, função-membro da classe string, 682, 718  
 data/hora universal, formato de, 378  
 Date, classe, 409, 421, 479, 480  
   definição da, com operadores de incremento sobrecarregados, 480  
   definição, 421  
   definições de função-membro e função friend da, 481  
   definições de função-membro, 422  
   exercício, 92  
   programa de teste da, 483  
 \_\_DATE\_\_, constante simbólica predefinida, 1026  
 DBMS (database management system), 671  
 De Morgan, leis de, 179  
 Debug, configuração, 1100  
 dec, manipulador de fluxo, 621, 621, 626, 630  
 decisão lógica, 3  
 decisão na UML, 170, 171  
 decisão, 2, 98, 99  
 declaração antecipada (forward declaration), 402, 402  
 declaração de classe antecipada (forward class declaration), 444, 444  
 declaração externa, 849  
 declaração, 31, 95  
   de uma função, 74  
 declarações  
   using, 38  
 declarar uma função-membro static const, 440

decoração de nome, 216  
 decrementar um ponteiro, 330, 331  
 decremento de uma variável de controle, 141  
 decremento prefixado, operador de, 121, 121  
 decritador, 724  
 default, caso, 155, 156, 192  
 #define NDEBUG, 1026  
 #define PI 3.14159, 1023  
 #define, diretiva de pré-processador, 377, 378, 596, 1022, 1024, 1026  
 Definição da classe Implementation, 443  
 Definição da classe Time modificada para permitir chamadas de função-membro em cascata, 432  
 definição, 141  
 Definições de função-membro da classe Time, 379  
 Definições de função-membro da classe Time, incluindo as funções-membro const, 414  
 Definições de função-membro da classe Time, incluindo um construtor que aceita argumentos, 388  
 definindo um construtor, 69  
 definindo uma classe com uma função-membro, 57  
 definir uma classe, 58  
 Deitel Buzz Online, boletim, 20  
 deitel@deitel.com, 3  
 deixar um loop, 179  
 del, elemento, 1074  
 delegação, 815  
 delete [] (desalocação dinâmica de array), 435  
 delete, comando de depurador, 1116  
 delete, operador, 430, 467, 585, 659, 800, 1011  
 delete, tipo de solicitação HTTP, 729  
 delimitador, 347, 352, 737  
   com valor padrão '\n', 618  
 delimitador-padrão, 619  
 demonstração da função utilitária, 386  
 demonstrando a atribuição e a concatenação de strings, 705  
 demonstrando a entrada a partir de um objeto istream, 721  
 demonstrando a função substr, 709  
 demonstrando as funções erase e replace, 714  
 demonstrando as funções string find, 713  
 demonstrando as funções string insert, 716  
 demonstrando as palavras-chave de operador, 969  
 demonstrando herança múltipla, 974  
 demonstrando o operador const\_cast, 965  
 demonstrando o template de classe Stack, 600, 602  
 demonstrando o uso de namespaces, 967  
 demonstrando os operadores .\* e ->\*, 973  
 demonstrando um membro de dados mutable, 971  
 Departamento de Defesa (DOD) dos Estados Unidos, 7  
 dependente de máquina, 5, 330, 440  
 depender de compilador, 314  
 Deposit, classe (estudo de caso ATM), 83, 85, 124, 229, 293, 294, 297, 401, 585, 586, 587  
 DepositSlot, classe (estudo de caso ATM), 83, 84, 85, 124, 229, 293, 294, 401

- depuração, recurso auxiliar na, 1025
- depurador (GNU C++)
  - comando break, **1114**
  - comando continue, **1115**
  - comando delete, **1116**
  - comando finish, **1120**
  - comando gdb, **1114**
  - comando help, **1115**
  - comando info break, **1116**
  - comando list, **1115**
  - comando next, **1120**
  - comando print, **1115**, 1117, 1118
  - comando quit, **1117**
  - comando run, **1114**
  - comando set, **1118**, 1118
  - comando step, **1119**
  - comando watch, **1121**, 1121
  - definição, **1112**
  - erros de lógica, 1112
  - inserir um ponto de interrupção, 1114
  - modo de interrupção, **1115**
  - opção de compilador -g, **1114**
  - ponto de interrupção, **1112**
  - suspender a execução do programa, 1118
- depurador (Visual C++ .NET)
  - barra indicadora marginal, **1100**
  - caixa de combinação *Solution Configurations*, **1100**
  - comando Continue, 1101, **1101**
  - comando Step Into, **1106**
  - comando Step Out, **1106**
  - comando Step Over, **1106**
  - configuração Debug, **1100**
  - definição, **1098**
  - erros de lógica, 1098
  - inserir um ponto de interrupção, 1100
  - janela Autos exibe estados de objetos, 1108
  - janela Locals, **1103**
  - janela Watch, **1103**, 1104
  - modo de interrupção, 1100, 1101
  - ponto de interrupção, **1098**
  - suspensão a execução do programa, 1103
- depurador, 1023, **1098**
- depurar, 6, 11, 195, 1065
- deque, classe, **892**, 901
  - contêiner de sequência, 911
  - função push\_front, **911**
- <deque>, arquivo de cabeçalho, 191, 895, **911**
- dequeue, **817**
- derivar uma classe de outra, **381**
- desalocar memória, 430, **431**, 435, 659, 800
- desassociar um fluxo de entrada de um fluxo de saída, 636
- desativar um ponto de interrupção, **1102**
- desempenho da classificação e da pesquisa de árvore binária, 845
- desempenho de pesquisa de lista, 845
- desempenho de pesquisa, 845
- desempenho, 7
- desempilhamento, **649**, 654, 655, 663
- desempilhar a pilha de chamadas de função, 654
- desencadear um nó de uma lista, 810, 811
- desenfileiramento, operação de, **442**, 442
- desenvolvimento de software orientado a componentes, 441
- desenvolvimento rápido de aplicativos (*rapid applications development* – RAD), **399**
- desfiguração de nome, **216**
  - para permitir linkagem segura para tipos, 218
- desigualdade, 969
  - operador de, (!=), 458
  - palavra-chave de operador de, 969
- deslocamento a partir do início de um arquivo, **675**
- deslocamento de bits, operadores de, 858
- deslocamento para a direita (>>), operador de, 451, 614, **853**, 853, 859, 886
- deslocamento para a direita de um valor com sinal é dependente de máquina, 859
- deslocamento para a esquerda (<<), operador de, 451, 614, **853**, 853, 857, 886
- deslocamento para a esquerda (<<=), operador de atribuição de, **860**
- deslocamento para um ponteiro, 332
- deslocamento, **581**
- deslocar um intervalo de números, **192**
- desreferência (\*), operador de, **314**
- desreferenciar
  - um iterador const, 897
  - um iterador posicionado fora de seu contêiner, 897
  - um iterador, 896, 900
  - um ponteiro nulo, 314
  - um ponteiro, **314**, 314, 316, 317, 320
- destrutor não virtual, **585**
- destrutor, **391**, 391, 515, 894
  - chamada de, na ordem inversa das chamadas de construtor, 392, 532
  - em uma classe derivada, 532
  - não recebe parâmetros e não retorna um valor, 391
  - sobrecarregar um, 391
- desvio incondicional goto, 1012
- desvio incondicional, 1012
- detalhes ‘segredos’ de implementação, 971
- determinar dinamicamente qual função utilizar, **557**
- determinar se uma string é um palíndromo, exercício de recursão, 228, 310
- developer.intel.com/software/products/compilers/cwin/index.htm, 21
- diagnósticos que auxiliam na depuração de programa, 192
- diagrama de atividades (UML), **46**, 96, 99, 100, 103, 104, 145, 146, 165, 170, 171, 172
  - instrução do...while, 151
  - instrução for, 145
  - instrução if, 99
  - instrução switch, 158
  - instruções em sequência, 96
- diagrama de atividades da estrutura de sequência, 96
- diagrama de atividades de instrução de seleção múltipla switch com instruções break, 158
- diagrama de atividades mais simples, 165, 166, 167
- diagrama de classes
  - na UML, **45**, **59**, 60, **83**, 85, 123, 125, 229, 399, 400, 401, 587, 588, 589
  - para o modelo do sistema ATM, 85, 86, 87
- diagrama de colaboração na UML, **46**, **294**
- diagrama de comunicação na UML, **46**, **294**, 294
- diagrama de estados do objeto ATM, 169
- diagrama de estados na UML, **169**
- diagrama de interação na UML, **294**
- diagrama de objetos na UML, **1057**
- diagrama de seqüências na UML, **46**, **294**
- diagrama elidido da UML, **83**
- diagramas de componentes na UML, **1057**
- diagramas de estados de máquina na UML, **46**, **169**
- diagramas de estrutura compostos na UML, **1057**
- diagramas de implantação na UML, **1057**
- diagramas de pacotes na UML, **1057**
- diagramas de sincronização na UML, **1058**
- diagramas de visão geral das interações na UML, **1057**
- Dica de desempenho, **6**
- Dica de portabilidade, **6**, 11
- Dica de prevenção de erro, **6**
- diferente de zero tratado como true, 164
- difference\_type, 895
- digitalizar imagens, 3
- dígito decimal, **669**
- dígito, 32, 346, 992
- Dígitos invertidos, exercício, 244
- dígitos significativos, 626
- direção de busca (*seek direction*), **675**
- direita para a esquerda, associatividade da, 40, 123
- diretivas condicionais de pré-processador, 1024, 1025
- diretivas de pré-processador, 10, **28**, 31
  - #define, **377**, 378, **1022**, 1024, 1026
  - #endif, **377**, 378, 1024, 1025
  - #if, 1024, 1025
  - #ifdef, **1024**
  - #ifndef, **377**, 378, **1024**
  - #include, 189, **1022**, 1022
  - #pragma, **1025**
- diretório de sistema de arquivos, 799
- diretório htdocs, **731**
- diretório virtual, **732**
- disco (XHTML), **1075**
- disco magnético, 669
- disco óptico, 669
- disco rígido, 3
- disco, 10, 836, 838
- dispositivo de armazenamento secundário, 10, **669**
- dispositivo de entrada, **3**
- dispositivo de fala, 1077
- dispositivo de saída, **3**
- distinção de letras maiúsculas e minúsculas, **32**
- DivideByZeroException, 650
- DivideByZeroException, definição da classe, 647
- divides, objeto de função, 953
- dividir para conquistar, abordagem, **182**, 183



dividir um vetor na classificação por intercalação, 788

divisão de inteiros, 35, 115

divisão por zero é indefinida, 441

divisão por zero, 110, 1009

divisão, 3, 35, 36

DNS (Domain Name System), 731

DNS, pesquisa, 731

do...while, instrução de repetição, 97, 150, 151, 169

documentar um programa, 27

documento de requisitos, 16, 41, 44, 45

dois maiores valores, 135

dois níveis de refinamento, 111

dois-pontos (:), 202, 421, 974, 1012

domínio de nível mais alto (*top-level domain* – TLD), 730

- cn, 731
- com, 731
- edu, 731
- et, 731
- om, 731
- org, 731
- us, 731

domínio, 730

double, tipo de dados, 111, 147, 148, 189, 1009

downcasting, 557, 562

dynamic\_cast, operador, 582, 661

**E**

E lógico (&&), operador, 161, 161, 179, 855

E lógico, 969

E sobre bits

- operador de atribuição, (&=), 860, 969
- operador, (&), 853, 853, 855, 886

E sobre bits, OU inclusivo sobre bits, OU exclusivo sobre bits e de complemento de bits, operadores, 855

E/S de alto nível, 613

E/S de vídeo, 614

E/S formatada, 613

E/S fortemente tipada (*type-safe I/O*), 612

E/S na memória, 719

E/S não formatada utilizando as funções-membro read, gcount e write, 620

E/S não formatada, 613, 613, 614, 620

EBCDIC (Extended Binary Coded Decimal Interchange Code), 350, 351

economizar memória, 200

edição, 9

editar um arquivo, 10

editor de textos, 345, 674, 1064

editor, 10

edu (domínio de nível mais alto), 731

efeito colateral, 201, 209, 210, 225, 596

eficiência de

- classificação por borbulhamento, 796
- classificação por inserção, 788
- classificação por intercalação, 791
- classificação por seleção, 787
- pesquisa binária, 785
- pesquisa linear, 782

elemento aninhado, 1066

elemento de um array, 251

elemento de uma string, 732

elemento divisor, 949

elemento fora do intervalo, 465, 478

elemento lista não ordenada (u1), 1075

elemento vazio (XHTML), 1072, 1072

elementos de nível de título (XHTML), 1066

elementos de nível de título

- h1, 1067
- h2, 1067
- h3, 1067
- h4, 1067
- h5, 1067
- h6, 1067

eleva a uma potência, 175, 184

eleva um inteiro a uma potência inteira, exercício de recursão, 227

#elif, 1024

eliminação de duplicatas, 827, 833

else oscilante, problema do, 101, 136, 137

emacs, 10

emacs, editor de textos, 1064

embaralhamento, algoritmo de, 850

embaralhar cartas, 336, 886

empacotador de pré-processador, 377, 377, 378

empilhamento, 99, 100, 169

empilhamento, regra de, 165

Employee, classe, 421, 422, 426

- arquivo de cabeçalho da, 567
- arquivo de implementação da, 567
- definição da, com um membro de dados static para monitorar número de objetos Employee na memória, 437
- definição da, para mostrar a composição, 423
- definições de função-membro da, 437
- definições de função-membro da, incluindo o construtor com uma lista de inicializadores de membro, 424
- exercício, 92
- programa de driver da hierarquia, 576

empresa de pedidos pelo correio, 178

empty, função-membro

- de contêineres, 894
- de priority\_queue, 923
- de queue, 921
- de stack, 920
- de string, 487, 710
- de um contêiner de sequência, 907

encapsulamento, 17, 66, 380, 397, 426

encriptador, 724

end line, 33

end, função-membro da classe string, 718

end, função-membro de contêineres de primeira classe, 896

end, função-membro de contêineres, 894

end, iterador, 899

endereço de loopback, 731

endereço Web, 730

endereço

- de um campo de bit, 862
- de uma estrutura, 848

#endif, diretiva de pré-processador, 377, 378, 1024, 1025

endl, manipulador de fluxo, 33, 116

enfileiramento, operação de, 442, 442

engenharia de software, xxvi, 57, 73, 74

- encapsulamento, 66
- funções set e get, 66
- ocultação de dados, 65, 66
- reutilização, 70, 73
- separar a interface da implementação, 73

enquete, 261, 263

enquete, programa de análise de, 262

enqueue, função, 817

Enter, tecla, 33, 155, 156

entrada a partir de strings na memória, realizar, 719

entrada de dados, 10

entrada de fluxo, 614, 616

entrada de strings na memória, 191

entrada de teclado, 113

entrada de uma string utilizando cin com extração de fluxo contrastada com a entrada utilizando cin.get, 618

entrada e saída de string, 845

entrada e saída, operações de, 96

entrada única/saída única, instrução de controle de, 98, 99, 165

entrada/saída (E/S), 183, 612

entrada/saída de objetos, 695

entrada/saída formatada, 680

entrada/saída, fluxo de, 28

entrada-padrão, 735, 1002

entradas ocultas, 1084

enum, palavra-chave, 197

enumeração, 197, 1022

EOF, 617, 619, 863

eof, função-membro de fluxo, 617, 634, 635, 636

eofbit de fluxo, 634

equação de linha reta, 36

equal, algoritmo STL, 901, 926, 926

equal\_range, algoritmo STL, 943, 944, 944

equal\_range, função de contêiner associativo, 915

equal\_to, objeto de função, 953

erase, função-membro

- da classe string, 714, 714
- de contêineres, 894, 907

erro de compilação, 29

erro de compilador, 29

erro de estouro, 440

erro de formato, 634

erro de linker, 965

erro de lógica fatal, 38, 110

erro de lógica, 38, 1098, 1112

erro de overflow aritmético, 662

Erro de programação comum, 6

erro de runtime, 10

erro de sintaxe, 29

erro de tempo de execução, 10

erro de underflow aritmético, 662

erro detectado em um construtor, 654

erro em tempo de compilação, 29

erro fatal de tempo de execução, 111

erro fatal, 10, 111, 366

erro síncrono, 650

erro, 10

off-by-one, **108**, 143, 144, 252  
 #error, diretiva de pré-processador, **1025**  
 escalares, **267**, 325  
 escalonamento, **192**  
 escalar, 256  
 escalonável, **256**  
 escapar no começo de um loop, 159  
 escopo de um identificador, 200, 201  
 escopo de uma constante simbólica ou macro, 1024  
 escopo global, 392, 966  
 escopo, **144**, 965, 966, 1005  
   de arquivo, **202**  
   de bloco, **202**  
   de classe, **202**  
   de função, **202**  
   de namespaces, **202**  
   de protótipo de função, **202**  
   exemplo de escopo, 203  
 escrutinar dados, 378  
 esgotar a memória, 222  
 espaçamento interno, 626, 628  
 espaçamento vertical, 98, 142  
 espaço ( ' ' ), 32  
 espaço em branco, caractere de, 98  
 espaço em disco, 657, 658, 673  
 espaço não pode ser alocado, 1012  
 espaço vazio, 682  
 espaços em branco, 178  
 espaços para preenchimento, 628  
 espaço-tempo, relação de troca, 688  
 especialização explícita de um template de classe, **605**  
 especialização na UML, **586**  
 especificação de exceção vazia, **653**  
 especificação de exceção, **653**  
 especificação de projeto, **45**  
 especificações de linkagem, **1016**, 1016  
 especificador de acesso, **58**, 64, 65, 399, 427  
   private, 64, 65  
   protected, 377  
   public, **58**, 64, 65  
   rótulo de, private:, 64  
   rótulo, public:, **58**  
 especificador puro, **563**  
 especificadores de classe de armazenamento, **200**  
   auto, **200**  
   extern, **200**  
   mutable, **200**  
   register, **200**  
   static, **200**  
 especificidades, 547  
 espiral, 223  
 esquerda para a direita, associatividade da, 40, 41, 123  
 estação de trabalho, **4**  
 estado consistente, **80**, 378, 387  
 estado de ação (UML), **96**, 96, 166, 167, 170  
   símbolo de, **96**  
 estado de erro de um fluxo, 617, 634, 635  
 estado de formatação, 620  
 estado de formato, **633**  
 estado de um objeto, 123, 169  
 estado final na UML, 170

estado final, **96**, 165  
 estado inicial na UML, **96**, 165, **169**, 170  
 estado na UML, **46**, 170  
 estado, **46**  
 estender a linguagem de programação básica, 441  
 estouro aritmético, **440**, 650  
 estouro de pilha, **205**  
 estouro, 1009  
 estrutura auto-referencial, **848**  
 estrutura de controle, 95, **96**  
 estrutura de dados bidimensional não-linear, 820  
 estrutura de dados dinâmica, 312, **799**  
 estrutura de dados linear, **801**, 820  
 estrutura de dados subjacente, 922  
 estrutura de dados, **251**, **799**, 891  
 estrutura de um sistema, **45**, 126, 169  
 estrutura funcional de um programa, 29  
 estrutura, **251**, 847, 848, 849, 1022  
   definição de, 847, 848, 860  
   membro, **847**  
   membros de, assumem o padrão de acesso  
     public, 847  
     nome, **847**, 848  
     operador de membro de, (.), 1014  
 estruturas de dados não-lineares, **801**  
 estruturas de dados pré-empacotadas, 799  
 estruturas de dados, xxxii, xxxiii  
 estruturas profundamente aninhadas, 165  
 esvaziar fluxos, 620  
 esvaziar o buffer de saída, 33  
 esvaziar o buffer, 636  
 esvaziar um fluxo, 1007  
 et (domínio de nível mais alto), 731  
 etapa de análise do ciclo de vida do software, 44  
 evento assíncrono, **650**  
 evento inesperado, 1009  
 evento, 169  
 evitar a repetição de código, 388  
 evitar vazamentos de memória, 659  
 exceção não listada na especificação de exceção, 653  
 exceção, **645**  
 exceções  
   bad\_alloc, **656**  
   bad\_cast, **661**  
   bad\_exception, **661**  
   bad\_typeid, 661  
   length\_error, **662**  
   logic\_error, **661**  
   out\_of\_bounds, 907  
   out\_of\_range, **662**, 705, 706, 908, 950  
   overflow\_error, 662  
   runtime\_error, **646**, 654, 661, 662  
   underflow\_error, 662  
 exception, classe, **646**, 661  
   função virtual what, **646**, 646  
 <exception>, arquivo de cabeçalho, 191, **646**, 653, 661  
 excluir memória alocada dinamicamente, 440  
 excluir um item de uma árvore binária, 828  
 excluir um registro de um arquivo, 695  
 exclusão de árvore binária, exercício de recursão, 228  
 .exe, extensão de arquivo, 734, 735

execução condicional de diretivas de pré-processador, **1022**  
 execução de um programa, 9, 10  
 execução sequencial, **95**  
 Exemplo de tratamento de exceções que lança exceções nas tentativas de divisão por zero, 647  
 exemplos e exercícios de recursão, 227, 228  
 Exemplos  
   A classe BasePlusCommissionEmployee representa um empregado que recebe um salário-base além de uma comissão, 511  
   Acessando funções-membro de um objeto por meio de cada tipo de handle de objeto, 383  
   Aleatorizando o programa de lançamento de dados, 196  
   Algoritmos de pesquisa e classificação básicos da Standard Library, 935  
   Algoritmos equal, mismatch e lexicographical\_compare, 926  
   Algoritmos matemáticos da Standard Library, 932  
   Algoritmos min e max, 948  
   Algoritmos swap, iter\_swap e swap\_ranges, 937  
   Alinhamento à esquerda e à direita com manipuladores de fluxo left e right, 627  
   Apontando um ponteiro de classe derivada para um objeto de classe básica, 555  
   Argumentos-padrão para uma função, 214  
   Arquivo de cabeçalho da classe BasePlusCommissionEmployee, 575  
   Arquivo de cabeçalho da classe CommissionEmployee, 572  
   Arquivo de cabeçalho da classe Employee, 567  
   Arquivo de cabeçalho da classe HourlyEmployee, 571  
   Arquivo de cabeçalho da classe SalariedEmployee, 568  
   Arquivo de implementação da classe BasePlusCommissionEmployee, 575  
   Arquivo de implementação da classe CommissionEmployee, 573  
   Arquivo de implementação da classe Employee, 567  
   Arquivo de implementação da classe HourlyEmployee, 571  
   Arquivo de implementação da classe SalariedEmployee, 569  
   Array de ponteiros para funções, 344  
   Arrays de caracteres processados como strings, 265  
   Atribuição-padrão de membro a membro, 398  
   Calculando a soma dos elementos de um array, 258  
   Cálculos de juros compostos com for, 148  
   Caracteres especiais HTML/XHTML, 1096  
   Chamadas de função-membro em cascata, 432, 434  
   Classe adaptadora priority\_queue da Standard Library, 923

- Classe adaptadora stack da Standard Library, 920
- Classe `BasePlusCommissionEmployee` que herda da classe `CommissionEmployee` mas não pode acessar diretamente os dados `private` da classe, 529
- Classe `BasePlusCommissionEmployee` que herda dados `protected` de `CommissionEmployee`, 523
- Classe `bitset` e o crivo de Eratóstenes, 951
- Classe `CommissionEmployee` com dados `protected`, 520
- Classe `CommissionEmployee` que representa um empregado que ganha uma porcentagem das vendas brutas, 507
- Classe `CommissionEmployee` utiliza funções-membro para manipular seus dados `private`, 527
- Classe `string` da biblioteca-padrão, 484
- Classe `Time` com funções-membro `const`, 414
- Classe `Time` contendo um construtor com argumentos-padrão, 387
- Classificação por seleção com passagem por referência, 326
- Colocando imagens em arquivos XHTML, 1071
- Comparando strings, 707
- Conjunto de chamadas recursivas à função `fibonacci`, 225
- Construtor com argumentos-padrão, 390
- `contact.html`, 1070, 1074
- Controlando a impressão de zeros finais e pontos de fração decimal para `doubles`, 627
- Convertendo strings em strings no estilo C e arrays de caracteres, 717
- Convertendo uma string em letras maiúsculas, 320
- Cópia de strings utilizando a notação de array e a notação de ponteiro, 335
- Copiando o diretório `examples`, xli
- Criando e manipulando um objeto `GradeBook` em que o nome do curso é limitado a 25 caracteres, 80
- Criando e percorrendo uma árvore binária, 825
- Criando um arquivo de acesso aleatório com 100 registros em branco sequencialmente, 685
- Criando um arquivo do lado do servidor para armazenar dados do usuário, 758
- Criando um arquivo sequencial, 672
- Dados de classe básica `protected` podem ser acessados a partir da classe derivada, 524
- Dados `private` da classe básica não podem ser acessados a partir da classe derivada, 516
- Definição da classe `Array` com operadores sobrecarregados, 459, 489
- Definição da classe `Complex`, 494
- Definição da classe `CreateAndDestroy`, 392
- Definição da classe `Date` com operadores de incremento sobrecarregados, 480
- Definição da classe `Date`, 421
- Definição da classe `DivideByZeroException`, 647
- Definição da classe `Employee` com um membro de dados `static` para monitorar o número de objetos `Employee` na memória, 437
- Definição da classe `Employee` para mostrar a composição, 423
- Definição da classe `GradeBook` contendo os protótipos de função que especificam a interface da classe, 75
- Definição da classe `Implementation`, 443
- Definição da classe `Integer`, 660
- Definição da classe `Interface`, 443
- Definição da classe `SalesPerson`, 384
- Definição da classe `String` com sobrecarga de operadores, 469
- Definição da classe `Time` modificada para permitir chamadas de função-membro em cascata, 432
- Definição da classe `Time`, 377
- Definição de função-membro da classe `Integer`, 660
- Definição do template de classe `List`, 803
- Definição do template de classe `ListNode`, 802
- Definição do template de classe `Queue`, 818
- Definição do template de classe `Stack`, 814
- Definição do template de classe `Tree`, 822
- Definição do template de classe `TreeNode`, 822
- Definições da função-membro e função `friend` da classe `Array`, 459
- Definições de função-membro da classe `Complex`, 495
- Definições de função-membro da classe `CreateAndDestroy`, 393
- Definições de função-membro da classe `Date`, 422
- Definições de função-membro da classe `Employee`, 437
- Definições de função-membro da classe `Employee`, incluindo construtor com uma lista de inicializadores de membro, 424
- Definições de função-membro da classe `Interface`, 444
- Definições de função-membro da classe `SalesPerson`, 385
- Definições de função-membro da classe `Time` incluindo um construtor que aceita argumentos, 388
- Definições de função-membro da classe `Time`, 379
- Definições de função-membro da classe `Time`, incluindo as funções-membro `const`, 414
- Definições de função-membro e função `friend` da classe `Date`, 481
- Definições de função-membro e função `friend` da classe `String`, 471
- Definições de função-membro para a classe `GradeBook` com uma função `set` que valida o comprimento do membro de dados `courseName`, 79
- Definindo a classe `GradeBook` com uma função-membro que aceita um parâmetro, 61
- Definindo a classe `GradeBook` com uma função-membro, criando um objeto `GradeBook` e chamando sua função-membro, 58
- Definindo e testando a classe `GradeBook` com um membro de dados e funções `set` e `get`, 63
- Demonstração da classe `GradeBook` depois de separar sua interface de sua implementação, 76
- Demonstração da função utilitária, 386
- Demonstrando a atribuição e a concatenação de strings, 705
- Demonstrando a entrada a partir de um objeto `istream`, 721
- Demonstrando a função `substr`, 709
- Demonstrando as funções da Standard Library `fill`, `fill_n`, `generate` e `generate_n`, 924
- Demonstrando as funções da Standard Library `remove`, `remove_if`, `remove_copy` e `remove_copy_if`, 928
- Demonstrando as funções da Standard Library `replace`, `replace_if`, `replace_copy` e `replace_copy_if`, 930
- Demonstrando as funções `erase` e `replace`, 714
- Demonstrando as funções `string find`, 713
- Demonstrando as funções `string insert`, 716
- Demonstrando as palavras-chave de operador, 969
- Demonstrando `copy_backward`, `merge`, `unique` e `reverse`, 938
- Demonstrando herança múltipla, 974
- Demonstrando `inplace_merge`, `unique_copy` e `reverse_copy`, 941
- Demonstrando `lower_bound`, `upper_bound` e `equal_range`, 944
- Demonstrando o operador `const_cast`, 965
- Demonstrando o template de classe `Stack`, 600, 602
- Demonstrando o uso de namespaces, 967
- Demonstrando os operadores `*` e `->`, 973
- Demonstrando `set_new_handler`, 658
- Demonstrando um membro de dados `mutable`, 971
- Desempilhamento, 655
- Desfiguração de nome para permitir linkagem segura para tipos, 218
- Documento XHTML contendo um formulário para postar dados para o servidor, 752
- Documento XHTML para ler informações de contato do usuário, 757



- E/S não formatada utilizando as funções-membro `read`, `gcount` e `write`, 620
- Entrada de uma string utilizando `cin` com extração de fluxo contrastada com a entrada utilizando `cin.get`, 618
- Estabelecendo links para outras páginas Web, 1068
- Estabelecendo links para um endereço de correio eletrônico, 1070
- Exemplo de escopo, 203
- Exemplo de tratamento de exceções que lança exceções nas tentativas de divisão por zero, 647
- Exemplos de herança, 503
- Formulário que inclui botões de opção e listas drop-down, 1089
- Formulário simples com campos ocultos e uma caixa de texto, 1084
- Friends podem acessar os membros `private` de uma classe, 427
- Função de conversão de string `atoi`, 870
- Função de conversão de strings `atof`, 869
- Função de conversão de strings `atoi`, 870
- Função de conversão de strings `strtod`, 871
- Função de conversão de strings `strtol`, 871
- Função de conversão de strings `strtoul`, 872
- Função de pesquisa de string `strchr`, 874
- Função de pesquisa de string `strcspn`, 875
- Função de pesquisa de string `strpbrk`, 875
- Função de pesquisa de string `strrchr`, 876
- Função de pesquisa de string `strspn`, 876
- Função de pesquisa de string `strstr`, 877
- Função de tratamento de memória `memchr`, 880
- Função de tratamento de memória `memcmp`, 879
- Função de tratamento de memória `memcpy`, 878
- Função de tratamento de memória `memmove`, 879
- Função de tratamento de memória `memset`, 880
- Função definida pelo usuário `maximum`, 185
- Função `inline` que calcula o volume de um cubo, 209
- Função-membro `flags` de `ios_base`, 633
- Função-membro `width` da classe `ios_base`, 623
- Funções de manipulação de elemento do template da classe `vector`, 906
- Funções de tratamento de caracteres `isdigit`, `isalpha`, `isalnum` e `isxdigit`, 864
- Funções de tratamento de caracteres `islower`, `isupper`, `tolower` e `toupper`, 866
- Funções de tratamento de caracteres `isspace`, `isctrl`, `ispunct`, `isprint` e `isgraph`, 867
- Funções não-`friend`/não-membro não podem acessar membros `private`, 428
- Funções que não aceitam argumentos, 208
- Funções `square` sobrecarregadas, 217
- Funções-membro `get`, `put` e `eof`, 617
- Gerando valores para serem colocados em elementos de um array, 256
- Gravando um cookie, 754
- Handler de portal interativo, 750
- Hierarquia de herança de `MembrosDaComunidade` da universidade, 504
- Implementando uma classe proxy, 445
- Imprimindo características de string, 711
- Imprimindo múltiplas linhas de texto com uma única instrução, 30
- Imprimindo o endereço armazenado em uma variável `char *`, 616
- Imprimindo o valor de uma união em ambos os tipos de dados de membro, 1014
- Imprimindo um inteiro com espaçamento interno e sinal de adição, 628
- Imprimindo um inteiro sem sinal em bits, 853
- Imprimindo uma linha de texto com múltiplas instruções, 30
- Imprimindo uma string um caractere por vez utilizando um ponteiro não constante para dados constantes, 321
- Incluindo a classe `GradeBook` a partir do arquivo `GradeBook.h` para utilizar em `main`, 72
- Inicialização de array `static` e inicialização de array automático, 265
- Inicializador de membro utilizado para inicializar uma constante de um tipo de dados predefinido, 417
- Inicializadores de objeto-membro, 425
- Inicializando arrays multidimensionais, 281
- Inicializando e utilizando corretamente uma variável constante, 257
- Inicializando elementos de um array como zeros e imprimindo o array, 254
- Inicializando os elementos de um array em sua declaração, 254
- Inicializando uma referência, 212
- Inserindo caracteres especiais em XHTML, 1074
- Inserindo dados de caracteres com a função-membro `cin.getline`, 619
- Instanciando múltiplos objetos da classe `GradeBook` e utilizando o construtor `GradeBook` para especificar o nome do curso quando cada objeto `GradeBook` é criado, 68
- Instrução `break` saindo de uma instrução `for`, 159
- Instrução `continue` que termina uma única iteração de uma instrução `for`, 160
- Instrução de repetição `do...while`, 150
- Instruções de controle aninhadas: problema dos resultados do exame, 119
- Inteiros deslocados e escalonados produzidos por `1 + rand() % 6`, 193
- Inteiros muito grandes, 499
- Iteradores de fluxo de entrada e de saída, 896
- Lançando um dado de seis lados 6.000.000 vezes, 193
- Lendo a entrada a partir de `QUERY_STRING`, 740
- Lendo cookies enviados a partir do computador do cliente, 756
- Lendo e imprimindo um arquivo seqüencial, 675
- Lendo um arquivo de acesso aleatório seqüencialmente, 688
- Listas aninhadas e ordenadas em XHTML, 1077
- Listas não ordenadas em XHTML, 1076
- Manipulações de array bidimensional, 282
- Manipulador de fluxo `showbase`, 630
- Manipuladores de fluxo `boolalpha` e `noboolalpha`, 632
- Manipuladores de fluxo `hex`, `oct`, `dec` e `setbase`, 621
- Manipuladores de fluxo não parametrizados definidos pelo usuário, 625
- Manipulando uma lista vinculada, 806
- Membro de dados `static` para monitorar o número de objetos de uma classe, 437, 439
- `new` lançando `bad_alloc` em caso de falha, 657
- Números complexos, 495
- Objeto `auto_ptr` gerencia a memória dinamicamente alocada, 661
- Objeto de função binária, 954
- Objetos `const` e funções-membro `const`, 416
- Operações `set` da Standard Library, 942
- Operador `sizeof` quando aplicado a um nome de array retorna o número de bytes no array, 328
- Operador `sizeof` utilizado para determinar tamanhos de tipo de dados padrão, 329
- Operador unário de resolução de escopo, 216
- Operadores de deslocamento de bits, 858
- Operadores de igualdade e relacionais, 39
- Operadores de inserção e de extração de fluxo sobrecarregados, 456
- Operadores de ponteiro `&` e `*`, 315
- Operadores E sobre bits, OU inclusivo sobre bits, OU exclusivo sobre bits e de complemento de bits, 855
- Passagem por referência com um argumento de ponteiro utilizado para elevar ao cubo o valor de uma variável, 317
- Passagem por valor utilizada para elevar o valor de uma variável ao cubo, 316
- Passando argumentos por valor e por referência, 211
- Passando arrays e elementos de array individuais a funções, 268
- Pesquisa linear de um array, 277
- Ponteiro `this` acessando implícita e explicitamente os membros de um objeto, 431
- Portal interativo para criar uma página Web protegida por senha, 749
- Precisão de valores de ponto flutuante, 622

Pré-incrementando e pós-incrementando, 122  
 Primeiro script CGI, 734  
 Programa de adição que exibe a soma de dois números, 31  
 Programa de análise de enquete, 262  
 Programa de *check out* para o exemplo do carrinho de compras, 774  
 Programa de classificação para múltiplos propósitos utilizando ponteiros de função, 342  
 Programa de consulta de crédito, 677  
 Programa de conta bancária, 689  
 Programa de driver da hierarquia de classes `Employee`, 576  
 Programa de impressão de gráfico de barras, 259  
 Programa de impressão de texto, 27  
 Programa de processamento de fila, 819  
 Programa de rolagem de dados utilizando um array em vez de `switch`, 260  
 Programa de teste da classe `Array`, 462  
 Programa de teste da classe `BasePlusCommissionEmployee`, 513  
 Programa de teste da classe `CommissionEmployee`, 509  
 Programa de teste da classe `Date`, 483  
 Programa de teste da classe `String`, 474  
 Programa que gera saída de uma página de login, 762  
 Programa simples de pilha, 815  
 Qualificador de tipo `const` aplicado a um parâmetro de array, 270  
 Recuperando variáveis de ambiente via função `getenv`, 738  
 Referência não inicializada produz um erro de sintaxe, 213  
 Referenciando elementos do array com o nome de array e com ponteiros, 333  
 Relançando uma exceção, 652  
 Repetição controlada por contador com a instrução `for`, 143  
 Retornando uma referência a um membro de dados `private`, 395  
 Script CGI que permite aos usuários comprar um livro, 768  
`set_new_handler` especificando a chamada de função quando `new` falha, 658  
 Simulação do jogo de dados *craps*, 197  
 Sinais definidos no cabeçalho `<csignal>`, 1009  
 Solução fatorial iterativa, 226  
 Somando inteiros com a instrução `for`, 147  
`strcat` e `strncat`, 349  
`strcmp` e `strncmp`, 351  
`strcpy` e `strncpy`, 349  
`strlen`, 353  
`strtok`, 352  
 Tabela XHTML complexa, 1082  
 Tabela XHTML, 1079  
 Template de classe `deque` da Standard Library, 912  
 Template de classe `list` da Standard Library, 908

Template de classe `map` da Standard Library, 918  
 Template de classe `multimap` da Standard Library, 917  
 Template de classe `multiset` da Standard Library, 913  
 Template de classe `set` da Standard Library, 916  
 Template de classe `Stack` com um objeto `List` composto, 817  
 Template de classe `vector` da Standard Library, 903  
 Template `vector` da C++ Standard Library, 289  
 Templates de classe adaptadora `queue` da Standard Library, 922  
 Tentando chamar uma função multiplamente herdada polimorficamente, 978  
 Tentando modificar dados por meio de um ponteiro não constante para dados constantes, 322  
 Tentando modificar um ponteiro constante para dados constantes, 325  
 Tentando modificar um ponteiro constante para dados não constantes, 323  
 Tentativa errônea de inicializar uma constante de um tipo de dados predefinido por atribuição, 419  
 Testando estados de erro, 635  
 Utilizando a função `swap` para trocar duas strings, 710  
 Utilizando a função-membro `fill` e o manipulador de fluxo `setfill` para alterar caractere de preenchimento para campos maiores que os valores sendo impressos, 629  
 Utilizando argumentos de linha de comando, 1006  
 Utilizando as classes básicas `virtual`, 980  
 Utilizando as funções `exit` e `atexit`, 1008  
 Utilizando as funções-membro `get`, `put` e `eof`, 617  
 Utilizando funções de template, 598  
 Utilizando funções Standard Library para realizar um `heapsort`, 946  
 Utilizando `goto`, 1013  
 Utilizando imagens como âncoras de link, 1072  
 Utilizando listas de argumentos de comprimento variável, 1004  
 Utilizando o manipulador de fluxo `uppercase`, 632  
 Utilizando o método `get` com um formulário XHTML, 743  
 Utilizando o método `post` com um formulário XHTML, 746  
 Utilizando o tratamento de sinal, 1010  
 Utilizando um iterador para gerar a saída de uma string, 718  
 Utilizando um objeto `ostream` dinamicamente alocado, 720  
 Utilizando um template de função, 219  
 Utilizando uma `union` anônima, 1015

Valores de ponto flutuante exibidos nos formatos-padrão, científico e fixo, 631  
 Variáveis `const` devem ser inicializadas, 257  
 exercícios sobre ponteiros, 366  
 Exercícios  
 Aprimorando a classe `Date`, 409  
 Aprimorando a classe `Rectangle`, 409  
 Aprimorando a classe `Time`, 408, 409  
 Classe `Account`, 91  
 Classe `Complex`, 408  
 Classe `Date`, 92  
 Classe `Employee`, 92  
 Classe `HugeInt`, 494  
 Classe `HugeInteger`, 409  
 Classe `Invoice`, 92  
 Classe `Rational`, 409  
 Classe `RationalNumber`, 500  
 Classe `Rectangle`, 409  
 Classe `TicTacToe`, 410  
 Classificação do tipo *bucket sort*, 796  
 Classificação por borbulhamento (*bubble sort*) aprimorada, 796  
 Classificação por borbulhamento (*bubble sort*), 303, 796  
 Combinando a classe `Time` e a classe `Date`, 409  
 Computadores na educação, 244, 245  
 Dígitos invertidos, 244  
 Gerenciador de tela polimórfico usando a hierarquia `Forma`, 594  
 Hierarquia de herança de `Account`, 544  
 Hierarquia de herança de `Package`, 544  
 Hierarquia de herança `Package`, 594  
 Hierarquia `Forma`, 594  
 Hipotenusa, 243  
 Instrução auxiliada por computador, 245  
 Jogo 'Adivinhe o número', 245  
 Modificação de sistema de folha de pagamento, 594  
 Modificando a classe `GradeBook`, 91  
 Números perfeitos, 244  
 Números primos, 244  
 Pesquisa binária recursiva, 797  
 Pesquisa linear recursiva, 797  
 Programa polimórfico de operações bancárias usando a hierarquia `Account`, 594  
*Quicksort*, 797  
 Retornando indicadores de erros das funções `set` da classe `Time`, 409  
 Série de Fibonacci, 246  
 Temperaturas Celsius e Fahrenheit, 244  
`exit`, função, 392, 658, 663, 674, 1007, 1007, 1008  
`EXIT_FAILURE`, 1007  
`EXIT_SUCCESS`, 1007  
`exp`, função, 184  
 expandir uma macro, 1023  
`explicit`, construtor, 488, 488, 489  
 'explosão' exponencial de chamadas, 225  
 expoente, 500  
 exponenciação, 37, 147  
 expressão algébrica, 35  
 expressão aritmética infix, 831

expressão condicional, **100**, 100, 378, 650  
 expressão de ação (UML), **96**, 99, 100, 104, 146, 151, 157, 170  
 expressão de coerção, 1024  
 expressão de controle (em um `switch`), **155**  
 expressão de ponteiro, 330, 332  
 expressão de tipo misto, **189**  
 expressão inteira constante, **151**, 157  
 expressão pós-fixa, 832, 843  
 expressão, 98, 100, 115, 116, 144, 145  
 Extended Binary Coded Decimal Interchange Code (EBCDIC), 350, 351  
 extensão de nome de arquivo, 10  
   `.h`, 70  
 extensão de sinal, 853  
 extensibilidade da STL, 893  
 extensibilidade do C++, 457  
 extensibilidade, 548  
 Extensible HyperText Markup Language (XHTML), xli, 728, 732, **1064**  
 extern "C", 1016  
 extern, especificador de classe de armazenamento, **200**, 201, 1005, 1006  
 extração de fluxo >> ('obtem de'), operador de, **31**, **33**, 40, 217, 218, 451, 455, 465, 614, 615, 616, 695

## F

F, sufixo de ponto flutuante, 1009  
 f, sufixo de ponto flutuante, 1009  
 fabs, função, 184  
 fail, função-membro, **634**  
 failbit de um fluxo, **617**, 620, 634, 673, 674  
 falar com um computador, 3  
 falhas não recuperáveis, 634  
 false, **98**, 100, 632  
 FAQs, 21, 1061  
 fase de compilação, 29  
 fase de edição, 10  
 fase de inicialização, 109  
 fase de processamento, 109  
 fases de um programa, 109  
 fator de escalonamento, **192**  
 fatorDeEscala, 195  
 fatores primos, 961  
 fatorial, 138, 139, 178, 221, 222, 223  
 faxina de terminação, **391**  
 'fazer sua pontuação', 197  
 fechar um fluxo, 1007  
 ferramenta de depuração, 1026  
 ferramenta de desenvolvimento de programas, 98, 111  
 Fibonacci, série de, 222, 223, 224, 225  
 FIFO (*first-in, first-out* – primeiro a entrar, primeiro a sair), **442**, 442, 817, 893, 911, 921  
 fila (*queue*), **442**, 442, **799**, 800, 801, 812, 817  
 fila com dupla terminação (*double-ended queue*), 911  
 fila cresce infinitamente, 833  
 fila de caixa em um supermercado, 817, 833  
 fila de espera, 442  
 filas em uma rede de computadores, 818  
 file (XHTML), valor de atributo, 742  
 \_\_FILE\_\_, constante simbólica, 1026  
 file-open, modo, **672**, 672, 673  
   ios::app, **672**, 673  
   ios::ate, 673  
   ios::binary, 673, **682**, 685  
   ios::in, 673, **675**  
   ios::out, **672**, 673  
   ios::trunc, 673  
 filho direito, **820**  
 filho esquerdo, **820**  
 filho, **820**  
 fill, algoritmo STL, 901, **924**  
 fill, função-membro de ostream, 626, **628**, 628, 629  
 fill\_n, algoritmo STL, 901, **924**  
 fim de um fluxo, 675  
 fim de uma lista, 845  
 fim de uma sequência, 934, 936  
 fim do arquivo (*end-of-file*), 155, 346, 347, 634, **671**, 671, 674, 1005  
   combinação de teclas, 674, 1002  
 'final de entrada de dados', 109  
 final de uma string, 800  
 find, algoritmo STL, 901, 934, **934**  
 find, função de contêiner associativo, **915**  
 find, função-membro da classe string, **712**, 712, 713, 716  
 find\_each, algoritmo STL, 901  
 find\_end, algoritmo STL, 901  
 find\_first\_not\_of, função-membro da classe string, **713**  
 find\_first\_of, algoritmo STL, 901  
 find\_first\_of, função-membro da classe string, **713**, 742  
 find\_if, algoritmo STL, 901, 934, **936**  
 find\_last\_of, função-membro da classe string, **713**  
 finish, comando de depurador, **1120**  
 first, membro de dados de pair, **915**  
 fixed, manipulador de fluxo, **116**, 626, **631**  
 flag, 838  
 flag, valor de, **109**  
 flags, função-membro de ios\_base, **633**  
 flip de bitset, **951**  
 float, 1009  
 float, tipo de dados, **111**, 190  
 <float.h>, arquivo de cabeçalho, 192  
 floor, função, 184, 242  
 fluxo de bytes, **613**  
 fluxo de caracteres, **28**  
 fluxo de controle de uma chamada de função virtual, 580  
 fluxo de controle na instrução `if...else`, 99  
 fluxo de controle, 103, 114  
 fluxo de entrada, 616, 618  
 fluxo de entrada/saída, **28**  
 fluxo de erro armazenado em buffer, 614  
 fluxo de erro padrão (cerr), 10  
 fluxo de erro-padrão não armazenado em buffer, 614  
 fluxo de saída, 905  
 fluxo de trabalho de parte de um sistema de software, **96**  
 fluxo de trabalho de um objeto na UML, 170  
 fmod, função, 184

fmtflags, tipo de dados, **633**  
 folha de estilo, **1065**  
 for, exemplos da instrução de repetição, 145  
 for, instrução de repetição, **97**, **143**, 143, 144, 169  
 for, instrução externa, 282  
 for\_each, algoritmo STL, 932, **934**  
 fora de escopo, 202  
 força bruta, computação baseada na, **179**  
 forçar um sinal de adição, 626  
 form (<form>...</form>), elemento XHTML, **741**, 741  
 form, elemento, 1084  
 forma de uma árvore, 845  
 Forma, hierarquia de classe, 504, 544  
   exercício, 594  
 formatação na memória, 719  
 formatação, 620  
 formato de ponto fixo, **116**  
 formato de saída de números de ponto flutuante, 631  
 formato do registro, 682  
 formato tabular, 253  
 formatos monetários, 192  
 formulário (XHTML), **728**, **1083**  
 formulário Web, 728, 729  
 formulário, 1064  
 fornecedor de software independente (*independent software vendor* – ISV), 6, 382, 540, 582  
 FORTRAN (FORMula TRANslator), **7**  
 Fortran, linguagem de programação (FORMula TRANslator), 1064  
 FQDN (*fully qualified domain name* — nome de domínio completamente qualificado), **730**  
 &frac14; (XHTML), caractere especial, 1074  
 frações, 500  
 frases com verbos no documento de requisitos, 229  
 free, 1011  
 frente de uma fila, 442  
 friend de template de classe, 606  
 friend de uma classe derivada, 974  
 friend, 454, 596, 606  
 friend, função, 426, 452, 458, 459, 504  
 friend, funções, não são funções-membro, 426  
 friend, funções, para aprimorar o desempenho, 426  
 Friends podem acessar os membros `private` de uma classe, 427  
 front, função-membro de contêineres de sequência, **902**, 905  
 front, função-membro de queue, **921**  
 front\_inserter, template de função, **938**  
 fstream, 671, 672, 685, 695  
 <fstream>, arquivo de cabeçalho, 191, 671  
 função auxiliar, **384**  
 função binária de comparação, 949  
 função binária, 949, **953**  
 função chamadora, **59**, 65, 183  
 função de acesso, **66**, **384**  
 função de classificação de array, 596  
 função de conversão de strings, **868**, 869  
   atof, 869

atoi, 870  
atol, 870  
strtod, 871  
strtol, 871  
strtol, 872  
função de Fibonacci recursiva, 227  
função de pesquisa de string  
  strchr, 874  
  strcspn, 875  
  strpbrk, 875  
  strrchr, 876  
  strspn, 876  
  strstr, 877  
função de terminação definida pelo programador, 653  
função definida pelo programador, 183  
função definida pelo usuário, 183, 185  
  maximum, 185  
função exponencial, 184  
função fatorial recursiva, 227  
função friend global, 457  
função geradora, 924  
função global a sobrecarregar um operador, 454  
função global, 184, 606  
função modificadora, 66  
função não-friend global, 452  
função operadora de coerção, 468  
função operadora, 454  
função predicado binária, 911, 926, 934, 936, 938, 940, 946, 949  
função predicado unária, 911, 928, 930  
função predicado, 384, 803, 911, 926, 928, 930, 932, 934, 936, 938, 940, 946, 949  
função recursiva, 221, 802  
função utilitária recursiva, 825  
função(ões), 6, 7, 10, 17, 28, 188, 189  
  argumento, 60  
  assinatura, 189, 216, 217  
  cabeçalho de, 59, 327  
  declaração (protótipo), 189  
  definição de, 62, 188, 202  
  escopo de, 202, 202  
  lista de parâmetros vazia, 206, 208  
  lista de parâmetros, 60, 61, 62  
  múltiplos parâmetros, 62  
  nome, 201, 341, 1007  
  parâmetro como uma variável local, 62  
  parâmetro, 60, 60, 61, 62  
  parênteses vazios, 59, 61  
  protótipo de, 74, 148, 188, 189, 202, 210, 318, 426, 1003, 1006, 1007, 1016  
  que chama a si mesma, 221  
  retorno de resultado, 65  
  sem argumentos, 208  
  sobrecarga de, 216, 612, 1003  
  variável local, 62  
função, corpo da, 59  
função-membro automaticamente colocada  
  inline, 379, 380  
função-membro definida dentro de uma definição de classe, 380  
função-membro não-const em um objeto const, 413

função-membro não-const em um objeto não-const, 413  
função-membro não-static, 429, 440  
função-membro, 17, 56, 57, 400  
  definindo, em um arquivo de código-fonte separado, 76  
  parâmetro, 60  
funções 'pré-empacotadas', 183  
funções da biblioteca de matemática, 148, 149, 184, 191, 238  
  ceil, 184  
  cos, 184  
  exp, 184  
  fabs, 184  
  floor, 184  
  fmod, 184  
  log, 184  
  log10, 184  
  pow, 184  
  sin, 184  
  sqrt, 184  
  tan, 184  
funções de adaptadores de contêiner  
  pop, 919  
  push, 919  
funções de biblioteca de entrada/saída, 192  
funções de contêineres associativos  
  count, 915  
  equal\_range, 915  
  find, 915  
  insert, 915, 918  
  lower\_bound, 915  
  upper\_bound, 915  
funções de memória da biblioteca de tratamento de strings, 877  
funções de pesquisa da biblioteca de tratamento de strings, 873  
funções de tratamento de caracteres isdigit, isalpha, isalnum e isxdigit, 864  
funções de tratamento de caracteres islower, isupper, tolower e toupper, 866  
funções de tratamento de caracteres isspace, iscntrl, ispunct, isprint e isgraph, 867  
funções de tratamento de caracteres, 863, 864  
Funções não-friend/não-membro não podem acessar membros private, 428  
funções para manipular dados em contêineres da biblioteca-padrão, 191  
funções-membro que não aceitam argumentos, 380  
<functional>, arquivo de cabeçalho, 191, 953, 953

## G

-g, opção de compilador de linha de comando, 1114  
gallery.yahoo.com, 1070  
gcount, função de istream, 620  
gdb, comando, 1114  
generalidades, 547  
generalização na UML, 586  
generate, algoritmo STL, 901, 924  
generate\_n, algoritmo STL, 901, 924  
Gerador de palavras cruzadas, 374

Gerador de palavras para números de telefone, 701  
gerando labirintos aleatoriamente, 228  
gerando labirintos de qualquer tamanho, 228  
gerando valores para serem colocados em elementos de um array, 256  
gerencia a memória dinamicamente alocada, 661  
Gerenciador de tela polimórfico usando a hierarquia Forma, exercício de, 594  
gerenciador de tela polimórfico, 547  
gerenciamento de memória dinâmico, 430  
get e set, funções, 66  
get, função, 66  
get, função-membro de istream, 617  
GET, método de HTTP, 732  
get, ponteiro, 675  
get, put e eof, funções-membro, 617  
get, tipo de solicitação HTTP, 728, 728, 729  
get, tipo de solicitação, 1084  
getenv, função, 737, 738, 740, 748  
getline, função de cin, 347, 618, 619  
getline, função do arquivo de cabeçalho string, 60, 66  
getline, função-membro da classe string, 705  
good, função de ios\_base, 635  
goodbit de fluxo, 635  
gosub, 844  
goto, eliminação do, 95  
goto, instrução incondicional no Simple, 836  
goto, instrução, 95, 95, 202, 1012  
grafia camelo, 58  
gráfico de barras, 178, 258, 259  
  programa de impressão de, 259  
gráfico de barras, informações em, 259  
gráfico de precedência e associatividade de operadores, 41  
gráfico, 178  
gráficos de tartaruga, 305  
Graphics Interchange Format (GIF), 1070  
gravação destrutiva, 34  
gravando dados aleatoriamente em um arquivo de acesso aleatório, 685  
greater, objeto de função, 953  
greater\_equal, objeto de função, 953  
grupo de campos relacionados, 670

## H

.h, extensão de nome de arquivo, 70  
h1, elemento de nível de título, 1067  
h2, elemento de nível de título, 1067  
h3, elemento de nível de título, 1067  
h4, elemento de nível de título, 1067  
h5, elemento de nível de título, 1067  
h6, elemento de nível de título, 1067  
handle de objeto, 383, 438  
handle em um objeto, 382  
handler de exceção, 649  
handler de formulário do lado do servidor, 729  
handler de formulário, 1084  
handler de interrupção, 347  
handler de sinal, 1010  
hardware, 2, 3, 5  
head (XHTML), elemento, 1065, 1065, 1066



head, tipo de solicitação HTTP, 729  
 heap, 431, 923, 945, 946, 948  
 heapsort, 922, 945  
 heapsort, algoritmo de classificação, 945  
 height, atributo do elemento img, 1072  
 help, comando de depurador, 1115  
 help-site.com, 1059  
 herança de implementação, 565  
 herança de interface, 565  
 herança em forma de losango, 978, 978  
 herança múltipla, 502, 503, 614, 973, 973, 974, 975, 976, 977, 978  
     demonstrando, 974  
 herança simples, 502  
 herança, 17, 377, 381, 502, 503, 504, 539, 540, 582, 585, 586, 587, 588, 589, 596, 843, 924  
     da classe básica virtual, 980  
     de implementação *versus* herança de interface, 565  
     exemplos de, 503  
     hierarquia de, 557, 564  
     hierarquia de, de MembrosDaComunidade da universidade, 504  
     relacionamentos de, das classes de E/S, 615, 671  
 herdar implementação, 594  
 herdar interface *versus* herdar implementação, 594  
 herdar interface, 587, 594  
 herdar membros de uma classe existente, 502  
 heurística, 307  
 Hewlett-Packard, 892  
 hex, manipulador de fluxo, 621, 626, 630  
 hexadecimal, 179, 871  
     inteiro, 314  
     notação, 615  
     sistema de numeração, (base 16), 616, 621, 626, 630, 871, 992  
     sistema de número, 864  
 hidden (type), valor do atributo, 742  
 hierarquia de categorias de iterador, 898  
 hierarquia de classes de E/S de fluxo, 615, 671  
 hierarquia de classes de exceções, 661  
 hierarquia de classes, 502, 564, 585, 843  
 hierarquia de dados, 669, 669  
 hierarquia de formas, 563  
 Hierarquia de herança de Package, exercício, 544  
 hierarquia de promoção para tipos de dados fundamentais, 190  
 hipertexto, 732  
 hipotenusa, 179, 243  
     exercício, 243  
 hora em que o arquivo-fonte é compilado, 1026  
 hora normal, 135  
 horas extras com 50% de gratificação, 135, 179  
 host, 730  
 hotwired.lycos.com/webmonkey/00/50/index2a.html, 1092  
 HourlyEmployee, arquivo de cabeçalho da classe, 571  
 HourlyEmployee, arquivo de implementação da classe, 571  
 hr (XHTML), elemento, 1074

href, atributo do elemento a, 1068  
 .htm (extensão de nome de arquivo XHTML), 1064  
 .html (extensão de nome de arquivo XHTML), 1064  
 HTML (Hypertext Markup Language), 728, 1064  
 html (XHTML), elemento, 1065  
 HTML, documento, xxxiv  
 .html, extensão de arquivo, 732  
 HTML, lista, xxxiv  
 HTML, tabela, xxxiv  
 HTML/XHTML, caracteres especiais, 1096  
 HTTP (Hypertext Transfer Protocol), 728, 729, 732  
 HTTP (versão 1.1), 733  
 HTTP, cabeçalho, 732, 733  
 HTTP, método, 732  
 HTTP, tipos de solicitação  
     delete, 729  
     head, 729  
     options, 729  
     put, 729  
     trace, 729  
 HTTP, transação, 735  
 HTTP\_COOKIE, variável de ambiente, 753  
 HTTPd (Apache), servidor Web, 732  
 httpd.apache.org/docs-2.0/, 735  
 HugeInt, classe, exercício, 494  
 HugeInteger, classe, exercício, 409  
 hyperlink de imagem, 1072  
 hyperlink, 732, 762, 1068  
 Hypertext Markup Language (HTML), 728, 1064  
 Hypertext Transfer Protocol (HTTP), 728, 732  
 hypotenuse, 238

## I

IBM Corporation, 4, 7  
 IBM Personal Computer, 4  
 Identificação de rede, 731  
 Identificação, guia na caixa de diálogo Rede, 731  
 identificador global, 965  
 identificador, 32, 97, 202  
     para um nome de variável, 200  
 #if, diretiva de pré-processador, 1024, 1025  
 if, instrução de uma única seleção, 37, 38, 40, 97, 98, 99, 167, 169  
     diagrama de atividades, 99  
 if...else, instrução de seleção dupla, 97, 99, 100, 167  
 #ifdef, diretiva de pré-processador, 1024  
 #ifndef, diretiva de pré-processador, 377, 378, 1024  
 ifstream, 671, 672, 674, 675, 687, 1005  
     função construtora, 674  
 ignore, função de istream, 455, 619  
 igual a, 38  
 igualdade e relacionais, operadores de, 38, 39  
 image (type), valor do atributo, 742  
 image/gif, tipo MIME, 733  
 imagem executável, 10  
 imagens em páginas Web, 1070  
 img (XHTML), elemento, 1072  
 img, elemento, 1072  
 impedir que arquivos de cabeçalho sejam incluídos mais de uma vez, 378  
 impedir que objetos de classe sejam copiados, 467  
 impedir que um objeto de uma classe seja atribuído a outra, 467  
 implementação de alterações em uma função-membro, 387  
 implementação de projeto, 229, 399, 400  
 implementação, 588, 589, 590  
 implementação, ocultamento dos detalhes da, 183, 440, 444  
 Implementando uma classe proxy, 445  
 implicitamente virtual, 557  
 imprecisão de números de ponto flutuante, 149  
 impressão de árvore, exercício de recursão, 228  
 impressora, 10, 613  
 imprimindo árvores, 835  
 imprimindo características de string, 711  
 imprimindo datas, 372  
 imprimindo múltiplas linhas de texto com uma única instrução, 30  
 imprimindo o endereço armazenado em uma variável char \*, 616  
 imprimindo o valor de uma união em ambos os tipos de dados de membro, 1014  
 imprimindo um inteiro com espaçamento interno e sinal de adição, 628  
 imprimindo um inteiro sem sinal em bits, 853  
 imprimindo uma linha de texto com múltiplas instruções, 30  
 imprimindo uma linha de texto, 27  
 Imprimindo uma string um caractere por vez utilizando um ponteiro não constante para dados constantes, 321  
 imprimir recursivamente uma lista de trás para frente, 833  
 imprimir um array, exercício de recursão, 228, 310  
 imprimir uma árvore binária em um formato de árvore bidimensional, 828  
 imprimir uma lista de trás para frente, 833  
 imprimir uma lista vinculada de trás para a frente, exercício de recursão, 228  
 imprimir uma lista vinculada de trás para frente, 802  
 imprimir uma string de trás para frente, exercício de recursão, 228, 310  
 inanição, 337  
 #include "filename", 1022  
 #include, diretiva de pré-processador, 189, 1022, 1022  
 includes, algoritmo STL, 940, 940  
 incluir um arquivo de cabeçalho múltiplas vezes, 377  
 inclusão circular, 404  
 incrementar o contador de instrução, 839  
 incrementar por um, 108  
 incrementar  
     um iterador, 900  
     um ponteiro, 330  
 incremento (++), operador de, 121, 479  
 incremento de uma variável de controle, 141, 144, 145, 146

incremento prefixado, operador de, [121](#), 121, 122  
 incremento, operadores de, 479  
 índice, [251](#)  
 indireção (\*), operador de, [314](#), 315  
 indireção, [312](#)  
 info break, comando de depurador, [1116](#)  
 informações de tipo em tempo de execução (*run-time type information* – RTTI), 191, [547](#), 582  
 informações sobre o tipo, 696  
 inicialização de array automático, 265, 266, 267  
 inicialização de array static e inicialização de array automática, 264  
 inicializador de membro para um membro de dados const, 418  
 inicializador de membro utilizado para inicializar uma constante de um tipo de dados predefinido, 417  
 inicializador de membro, [416](#), 416, 417, 418, 466, 476  
 inicializador, [253](#)  
 inicializadores de objeto-membro, 425  
 inicializando arrays multidimensionais, 281  
 inicializando e utilizando corretamente uma variável constante, 257  
 inicializando elementos de um array como zeros e imprimindo o array, 254  
 inicializando os elementos de um array em sua declaração, 254  
 inicializando uma referência, 212  
 inicializar com uma instrução de atribuição, 418  
 ‘inicializar duplamente’ objetos-membro, 426  
 inicializar em um estado consistente, 387  
 inicializar ponteiro como 0 (nulo), 803  
 inicializar um ponteiro declarado como const, 324  
 inicializar um ponteiro, 313  
 inicializar uma constante de um tipo de dados predefinido, 417  
 início de um arquivo, 675  
 início de um fluxo, 675  
 inline, função, [208](#), 208, 209, 384, 454, 468, 478, 845, 920, 922, 923, 1023, 1024  
     calcular o volume de um cubo, 209  
 inline, palavra-chave, [208](#), 379  
 inner\_product, algoritmo STL, 901  
 inOrderTraversal, 825, 827  
 inplace\_merge, algoritmo STL, 940, [940](#), 941  
 input (XHTML), elemento, 1084, 1086  
 input, comando Simple, 835, 836  
 input, elemento XHTML, 742  
 inserção de árvore binária, exemplo de recursão, 228  
 inserção de fluxo << (‘insere em’), operador de, [28](#), [29](#), 30, 33, 34, 217, 218, 451, 455, 465, 614, 615, 674, 974  
 inserção na parte de trás de um vector, 902  
 inserção, 442, 799  
 inserindo dados de caracteres com a função-membro cin.getline, 619  
 inserir em uma pilha, [204](#)  
 inserir linha de texto em um array, 347  
 inserir um ponto de interrupção, 1100  
 inserir uma linha de texto, 618

insert, função de contêiner associativo, [915](#), 918  
 insert, função-membro da classe string, 716, [717](#), 717  
 insert, função-membro de contêiner de sequência, [907](#)  
 instância de uma classe, 64  
 Instrução auxiliada por computador, exercício, 245  
 instrução branch zero do SML, 839  
 instrução composta, [40](#), [102](#)  
 instrução de atribuição, 33, 122  
 instrução de controle aninhada, [116](#), 165, 169  
     for, 260, 282, 286  
     if...else, [100](#), 101, 102  
     regra de aninhamento, [165](#)  
 instrução de controle, [94](#), 98, 99, 100  
     aninhamento, [98](#), 99, [116](#)  
     do...while, [97](#), 150, 151, 169  
     empilhamento, [98](#), 99, 165  
     for, [97](#), [143](#), 143, 144, 169  
     if, [37](#), 40, 167, 169  
     if...else aninhada, 101, 102  
     if...else, 167  
     instrução de repetição, 98  
     instrução de seleção, 98  
     instrução de sequência, 98  
     switch, 151, 156, 167  
     while, [97](#), 114, 141, 142, 150, 169  
 instrução de loop, [97](#), [103](#)  
 instrução de saída compilada condicionalmente, 1025  
 instrução de seleção dupla, [97](#), 151, 167  
 instrução de seleção múltipla, [97](#), 151  
 instrução de uma única seleção if, [97](#), 98, 101, 151  
 instrução estendida por várias linhas, 40  
 instrução executável, 33, [95](#)  
 instrução ilegal, [1009](#)  
 instrução nula (;), [102](#)  
 instrução vazia, [102](#)  
 instrução, 10, 28, [28](#), 59  
 instruções  
     break, 155, 156, 159, 179, 180  
     continue, 159, 179, 180  
     do...while, 150, 151, 169  
     for, [143](#), 143, 144, 169  
     if, [37](#), 38, 40, 167, 169  
     if...else, 167  
     return, 29  
     switch, 151, 156, 167  
     while, 141, 142, 150, 169  
 int &, 210  
 int, 28, 32, 189  
 int, operandos, promovidos a valores double, 115  
 Integer, definição da classe, 660  
 Integer, definição de função-membro da classe, 660  
 integerPower, 243  
 IntegerSet, classe, 449  
 integridade de uma estrutura de dados interna, 442  
 inteiro binário, 138

inteiro encriptado, 138  
 inteiro ímpar, 177  
 inteiro sem sinal em bits, 853, 854  
 inteiro, 28, [31](#), 138  
 inteiros aleatórios no intervalo de 1 a 6, 192  
 inteiros deslocados e escalonados produzidos por 1 + rand() % 6, 193  
 inteiros deslocados e escalonados, 193  
 Inteiros muito grandes, exercício, 499  
 inteiros prefixados com 0 (octal), 630  
 inteiros prefixados com 0x ou 0X (hexadecimal), 630  
 interações entre objetos, 292, 294  
 intercambialidade de arrays e ponteiros, 333  
 interceptar evento inesperado, [1009](#)  
 interface com o usuário, 730  
 interface de uma classe, 73, [74](#)  
 interface, [17](#), [73](#), 74, 547  
     herança de, [565](#)  
 Interface, definição da classe, 443  
 Interface, definições de função-membro da classe, 444  
 internal, manipulador de fluxo, 365, 626, [626](#)  
 International Organization for Standardization (ISO), 2, 6, 441  
 International Standards Organization (ISO), 2, 6  
 Internet Explorer (IE), 1064, 1072  
 Internet Explorer, 1064, 1072  
 Internet, [4](#), 5, 11, 12  
 interpretador, [5](#)  
 interrupção, [1009](#)  
 intervalo arbitrário de subscritos, 441  
 intervalo, [896](#), 932, 934  
 intervalos de inteiro aleatório, 833  
 intToDouble, 238  
 invalid\_argument, classe, [662](#)  
 invalid\_argument, exceção, 908  
 inverter a ordem dos bits em um inteiro unsigned, 886  
 invocando funções polimorficamente em uma classe derivada, 978  
 invocar uma função, 183  
 invocar uma função-membro não-const em um objeto const, 413  
 Invoice, classe, exercício, 92  
 <iomanip.h>, arquivo de cabeçalho, 191  
 <iomanip>, arquivo de cabeçalho, 116, 191, 614, 621, 1022  
 ios::app, modo file-open, [672](#), 673  
 ios::ate, modo file-open, 673  
 ios::beg, direção de busca, [675](#)  
 ios::binary, modo file-open, 673, [682](#), 685  
 ios::cur, direção de busca, [675](#)  
 ios::end, direção de busca, [675](#)  
 ios::in, modo file-open, 673, [675](#), 695  
 ios::out, modo file-open, [672](#), 673, 695  
 ios::trunc, modo file-open, 673  
 ios\_base, classe básica, 622  
 ios\_base, classe, 634  
     função precision, [621](#)  
     função-membro width, [622](#)  
 <iostream.h>, arquivo de cabeçalho, 191  
 <iostream>, arquivo de cabeçalho, 27, [28](#), 155, 191, 614, 671, 966, 1022

IP (Internet Protocol), endereço, **731**  
irmão, **820**  
isalnum, função, 864, **864**  
isalpha, função, 864, **864**  
iscntrl, função, 864, **866**  
isdigit, função, 864, **864**  
isgraph, função, 864, **866**  
islower, função, **321**, 864, **866**  
ISO, 2  
ISP (*Internet service provider* — provedores de serviço da Internet), 1084  
isprint, função, 864, **866**  
ispunct, função, 864, **866**  
isspace, função, 864, **866**  
istream, 614, 675, 681, 687, 695, 719  
  ignore, função-membro, 455  
istream, classe  
  função peek, **619**  
  função seekg, **675**  
  função tellg, **677**  
istream\_iterator, **896**, 896  
istreamstringstream, **719**, 721  
isupper, função, 864, **866**  
ISV, 582  
isxdigit, função, 864, **864**  
iter\_swap, algoritmo STL, 901, 937, **938**  
iteração, 104, 226, 226  
  de um loop, **104**  
iterador bidirecional, 897, 898, 902, 908, 913, 915, 938, 940, 949  
  operações, 900  
iterador de acesso aleatório, **783**, 897, 898, 902, 905, 911, 913, 926, 932, 936, 940, 946, 948, 949  
iterador de entrada, 898, 899, 900, 926, 928, 932, 934, 938, 940, 949  
iterador de fluxo de entrada, 896  
iterador de leitura direta, 897, 898, 902, 932, 936, 938, 949  
iterador de saída, 898, 899, 900, 924, 932, 940, 949  
iterador(a), **442**, 564, 596, 718, **892**  
  apontando para o primeiro elemento após o fim do contêiner, 896  
  apontando para o primeiro elemento do contêiner, 896  
  classe, 426, **564**  
  objeto, **442**  
  operação ++, 896  
  operação, 900  
iteradores de acesso aleatório, operações com, 900  
Iteradores de fluxo de entrada e de saída, 896  
iteradores de leitura direta, operações com, 900  
iterar, **108**  
iterator, 718, 894, 895, 896, 899, 915  
<iterator>, arquivo de cabeçalho, 191, 938, 940

## J

Jacobson, Ivar, 18, 19  
Jacopini, G., 95, 96, 167  
janela ativa, 1101  
japonês, 441

Java, **7**, 8  
Jogo 'Adivinhe o número', exercício, 245  
jogo da força, 725  
jogo de azar, 196  
jogo de dados *craps*, 197, 198, 199  
jogo de dados, 196, 197  
jogo, 190  
jogos de cartas, 336  
Joint Photographic Experts Group (JPEG), 1070  
juros compostos, 147, 178, 180  
  cálculo de, com for, 147, 148  
juros em depósito, 180

## K

Keypad, classe (estudo de caso ATM), 83, 84, 85, 229, 293, 295, 401  
KIS ('keep it simple', 'mantenha a coisa simples'), 11  
Koenig, Andrew, 645

## L

L, sufixo de inteiro, 1009  
l, sufixo de inteiro, 1009  
L, sufixo de ponto flutuante, 1009  
l, sufixo de ponto flutuante, 1009  
label (XHTML), elemento, 1086  
lado esquerdo de uma atribuição, 164, 251, 395, 465  
lados de um triângulo reto, 138  
lados de um triângulo, 138  
Lady Ada Lovelace, 7  
lançando dois dados, 197, 303, 304  
lançando um dado de seis faces 6.000.000 vezes, 192, 193, 194  
lançando um dado, 192, 193, 194  
lançar exceções derivadas de exceções-padrão, 662  
lançar exceções não derivadas de exceções-padrão, 662  
lançar exceções-padrão, 662  
lançar um int, 650  
lançar uma exceção, **647**  
lançar uma expressão condicional, 650  
largura configurada implicitamente como 0, 623  
largura de campo, **149**, 253, 620, 622  
largura de intervalo de números aleatórios, 195  
largura de um campo de bit, **861**  
largura para altura, relação de, 1072  
largura, configuração de, 622, 623  
latim de porco, 371  
layout de memória não contígua de um deque, 911  
Lee, Meng, 892  
left value, 165  
left, manipulador de fluxo, **149**, 626, **626**  
legibilidade, 98, 117, 118, 1065  
leitura de caracteres com getline, 60  
leitura não destrutiva, **34**  
Lendo e imprimindo um arquivo seqüencial, 675  
Lendo um arquivo de acesso aleatório seqüencialmente, 688  
length, função-membro da classe string, 79, **80**, **705**  
length\_error, exceção, **662**, 710, 908

ler dados seqüencialmente de um arquivo, 674  
less, objeto de função, 953  
less\_equal, objeto de função, 953  
less<double>, 915  
less<int>, **913**, 913, 915  
letra maiúscula, 32, 54, 191, 320, 864, 866  
letra minúscula, 32, 54, 98, 191, 321, 864, 866  
letra, **669**  
lexicographical\_compare, algoritmo STL, 926, **926**  
li (XHTML), elemento, **1075**  
liberar memória alocada dinamicamente, 467  
liberar memória, 800  
licenciamento de classes, 399  
LIFO (*last-in, first-out* — último a entrar, primeiro a sair), ordem, 204, 599, 602, 813, 893, 919  
limerick, 371  
limite de crédito em uma conta corrente, 134  
limite de uma unidade de armazenamento, 862  
limites de palavra dupla, 848  
limites de palavra, 848  
limites de tamanho inteiros, 192  
limites de tipos de dados numéricos, 192  
<limits.h>, arquivo de cabeçalho, 192  
<limits>, arquivo de cabeçalho, 192  
\_\_LINE\_\_, constante simbólica predefinida, 1026  
linguagem assembly, **5**  
linguagem de alto nível, **5**  
linguagem de máquina, **5**, 200, 839  
linguagem de marcação, xxxi, xxxiv, **1064**  
linguagem de programação procedural, **17**  
linguagem extensível, **19**, **59**, 222, 263, 441  
linguagem natural de um computador, 5  
linguagem orientada a objetos, 8, **17**  
linguagem portátil, 11  
linha da vida de um objeto em um diagrama de seqüências UML, **295**  
linha de base (XHTML), **1081**  
linha de comando, argumento de, **336**, 1005  
linha de comando, 1002  
linha de comunicação com um arquivo, 673, 674  
linha de tabela (tabela XHTML), **1081**  
linha de texto, 618  
linha em branco, 33, 111  
linha horizontal (XHTML), **1074**, 1074  
linha horizontal, xxxiv  
linha pontilhada, **96**  
linha reta, **35**, 36  
linha, 36  
linhas, **279**  
linkagem externa, **1007**  
linkagem fortemente tipada, 1016  
linkagem interna, **1007**  
linkagem segura para tipos (*type-safe linkage*), **216**  
linkagem, 9, **200**, 966  
linker, 10, 1005, 1007  
links.html, 1068  
links2.html, 1076  
Linux, 4, 731  
  shell no, 11  
list, comando de depurador, **1115**  
list, contêiner de seqüência, 908  
List, definição do template de classe, 803



- List, template de classe, 803, 814, 815, 818
- list, template de classe, **892**, 901, 919
  - assign, **911**
  - merge, 911
  - pop\_back, 911
  - pop\_front, **911**
  - push\_front, **908**
  - remove, **911**
  - sort, **911**
  - splice, **911**
  - swap, **911**
  - unique, **911**
- List< STACKTYPE >, 815
- <list>, arquivo de cabeçalho, 191, 895, **908**
- lista aninhada, 1077
- lista circular duplamente vinculada, **813**
- lista circular simplesmente vinculada, **812**
- lista de argumentos de comprimento variável, 1004
- lista de inicializadores de membro, **417**, 421, 422, 424, 974
- lista de parâmetros de template, **219**
- lista de parâmetros vazia, 206
- lista de parâmetros, **60**, 62, 69
- lista duplamente vinculada, **812**, 893, 908
- lista indexada, 845
- lista inicializadora, **253**, 255, 346
- lista ordenada, 1077
- lista separada por vírgulas, **32**, 40, 144, 313
  - de classes básicas, 974
  - de parâmetros, 188
- lista simplesmente vinculada, **812**
- lista vinculada, 442, **799**, 800, 801, 802, 806, 812
- lista, 800
- ListNode, definição do template de classe, 802
- literal, **33**
- live-code (código ativo), abordagem, **2**
- <locale>, arquivo de cabeçalho, 192
- localhost, **731**, 734, 735
- localização temporária, 840
- localizando strings e caracteres em uma string, 712
- localizando, xl
- Locals, janela, **1103**
- localtime, função, 734, **735**
- Location, cabeçalho, **748**
- log, função, 184
- log10, função, 184
- log, *n*, níveis, em uma árvore de pesquisa binária com *n* elementos, 828
- logaritmo natural, 184
- logaritmo, 184
- logic\_error, exceção, **661**
- lógica do negócio, **730**
  - camada da, **730**
- logical\_and, objeto de função, 953
- logical\_not, objeto de função, 953
- logical\_or, objeto de função, 953
- Logo, linguagem, 305
- long double, tipo de dados, 190, 1009
- long int, tipo de dados, 158, 190, 222
- long, tipo de dados, 158, 189, 1009
- loop controlado por sentinela, 836
- loop de retardo, **145**
- loop infinito, 103, 115, 138, 145, 222
- loop, **97**, **103**, 103, 104, 110
- Lord Byron, 7
- losango (UML), **96**, 98
- losango, 53, 179
- losangos sólidos (representando composição) na UML, **84**, 85
- Lovelace, Ada, 7
- lower\_bound, algoritmo STL, 943, **943**, 944
- lower\_bound, função de contêiner associativo, **915**
- &lt; (XHTML), caractere especial, 1074
- lvalue ('left value'), **165**, 212, 251, 314, 358, 395, 465, 468, 478, 912
- lvalue modificável, **292**, 465, 468, 487
- lvalue não modificável, **292**, 487
- lvalue, **292**
- lvalues como rvalues, 165
- M**
- m* por *n*, array, **279**
- Mac OS X, 4
- Macintosh, 623
- macro, 190, **596**, 596, 1003, **1022**
  - argumento, 1023
  - definição, 1025, 1026
  - expansão, 1024
  - identificador, **1023**
- macros de cabeçalho <cstdlib>, 1003
- magnitude alinhada à direita, 626
- magnitude, 626
- mailto:, URL, **1068**
- main, 28, 31, 1007
- mainframe, **2**
- maior elemento de uma coleção, 494
- maior que ou igual a, operador, 38
- maior que, operador, 38
- make, utilitário, **1007**, 1007
- make\_heap, algoritmo STL, **946**
- Makefile, **1007**
- malloc, 1011, 1012
- manipulação de bits, 669, 852, 853
- manipulação de caractere, 183
- manipulação de ponteiro perigosa, 579
- manipulação de ponteiro, 579, 799
- manipulação de strings de caracteres, 863
- manipulações de arrays bidimensionais, 286
- manipulador de fluxo não parametrizado, **116**
- manipulador de fluxo parametrizado, **116**, 149, **614**, 621, 624, 675
- manipulador(es) de fluxo, **33**, **116**, 149, **620**, 624, 625, 675
  - boolalpha e noboolalpha, 632
  - boolalpha, **163**, 632
  - dec, **621**
  - endl (end line), 33, 34
  - endl, 33, 34, 116
  - fixed, **116**, **631**
  - hex, **621**
  - internal, 626
  - left, **149**, 626, **626**, 627
  - noboolalpha, **632**
  - noshowbase, 626, **630**
  - noshowpoint, 626, **626**
- noshowpos, 365, 626, **626**
- noskipws, 626
- nouppercase, 626, **631**
- oct, 621, **621**, 626, 630
- right, **149**, **626**, 627
- scientific, **631**
- setbase, **621**, 621
- setfill, 365, **378**, 626, **628**, 628, 629
- setprecision, **116**, 149, 621, 622
- setw, **149**, 253, 347, 455, 623, 626
- showbase, **630**, 630
- showpoint, **116**, 626
- showpos, 626, **626**
- manipulador, 149, 671
- manipuladores de fluxo de estado de formato, 626
- Manipuladores de fluxo não parametrizados definidos pelo usuário, 625
- manipulando uma lista vinculada, 806
- manipular caracteres individuais, 863
- manutenção de software, 9
- map, contêiner associativo, 918
- <map>, arquivo de cabeçalho, 191, 895, **915**, 918
- mapeamento de um para muitos, 893
- mapeamento de um para um, 893, **918**
- Máquina Analítica, 7
- marcação, 732
- marcador de visibilidade na UML, **400**
- máscara, **854**
- <math.h>, arquivo de cabeçalho, 191
- max, algoritmo STL, **948**
- max\_element, algoritmo STL, 932, **934**
- max\_size, função-membro de contêineres, 894
- max\_size, função-membro de uma string, **710**
- maxheap, **945**
- máximo divisor comum (MDC), 244, 247
- máximo divisor comum recursivo, 227
- maximum, função, 185
- maxlength, atributo do elemento input, **1086**
- mdc, 247
- mecanismos de proteção, 399
- média aritmética, 36
- média áurea, **223**
- média da classe em um questionário, 104
- média da classe, generalização do problema da, 109
- média da classe, problema de, 104, 110, 111
- média de notas, 178
- média, 36, 104, 110, 111, 177
- meia-palavra, 845, 848
- melhorar o desempenho da classificação por borbulhamento, 796
- membro de dados private, atribuir o valor de um, 66
- membro de dados, 17, 57, **62**, 63, 64, 378, 400
  - private, 64, 65
- membros de classe assumem o padrão de acesso private, 847
- memchr, função de tratamento de memória, 880
- memchr, função, 877, **879**, 880
- memcmp, função de tratamento de memória, 879
- memcmp, função, 877, **878**
- memcpy, função de tratamento de memória, 878
- memcpy, função, 877, **878**, 878

memmove, função de tratamento de memória, 879  
 memmove, função, 877, **878**, 879  
 memória alocada dinamicamente, 398, 440, 466, 469, 477, 585, 659  
   array de caracteres, 426  
   array de inteiros, 465, 1019  
   para um array, 1011  
 memória básica, 3  
 memória dinâmica, 659  
 memória não alocada, 1012  
 memória principal, **3**  
 memória virtual, 657, 658, 800  
 memória, **3**, 10, 31, 34, 200, 201  
 memória, consumo de, 579  
 memória, endereço de, 312, 616  
 memória, posição na, 34, 35, 105  
 memória, vazamento de, xxxiii, **435**, 659, 718, 892  
   evitar, 659  
 <memory>, arquivo de cabeçalho, 191, **659**  
 memset, função de tratamento de memória, 880  
 memset, função, 877, **880**  
 menor de vários inteiros, 178  
 menor que ou igual a, operador, 38  
 menor que, operador, 38, 895, 896  
 mensagem (envio para um objeto), **57**  
 mensagem aninhada na UML, **295**  
 mensagem de erro, 10  
 mensagem de retorno na UML, 295  
 mensagem de solicitação, 636  
 mensagem na UML, **292**, 294, 295, 296  
 mensagem, 17, **28**, 451  
 merge, algoritmo STL, 938, **938**  
 merge, função-membro de `list`, **911**  
 mesclar dois objetos ordenados de lista, 831  
 method (XHTML), atributo, **741**, 1084  
 method = "get" (XHTML), 1084  
 method = "post" (XHTML), 1084  
 método (função), **183**  
 método de solicitação, **728**  
 método, 17  
 Meus locais de rede, 731  
 MFC, 8  
 Microsoft MFC (Microsoft Foundation Classes), 8  
 Microsoft Visual C++, 10, 956, 968, 1007  
   home page, 20, 1061  
 Microsoft Windows, 155  
 MIME (Multipurpose Internet Mail Extensions), tipo, **733**, **733**  
   image/gif, **733**  
   text/html, **733**  
   text/plain, **733**  
 min, algoritmo STL, **948**  
 min\_element, algoritmo STL, 932, **932**  
 minus, objeto de função, 953  
 mismatch, algoritmo STL, 901, 926, **926**  
 Mistério, exercício de recursão, 227  
 modelo de ação/decisão de programação, 99, 100  
 modelo de retomada do tratamento de exceções, **649**  
 modelo de terminação do tratamento de exceções, **649**  
 modelo de um sistema de software, 126, 587

modelo em cascata, **44**  
 modelo iterativo, **44**  
 modelo, 364  
 Modificação de sistema de folha de pagamento, exercício, 594  
 modificações no compilador de Simple, 843  
 Modificando a classe GradeBook, exercício, 91  
 modificar um ponteiro constante, 323  
 modificável, **292**  
 modo de acesso-padrão de membros de classe é `private`, 65  
 modo de interrupção, **1100**, 1100, 1101, **1115**  
 modularizar um programa com funções, 183  
 módulo de objeto, 382  
 módulo, 35, 36  
   operador, (%), **35**, 53, 54, 138, 192, 195  
 modulus, objeto de função, 953  
 moeda, lançamento de, 192, 244  
 mouse, 3  
 multimap, contêiner associativo, 915  
 multiplicação, 35, 36  
 multiplicidade, **84**, 84  
 multiplies, objeto de função, 953  
 múltiplos parâmetros para uma função, 62  
 multiprocessador, **4**  
 multiprogramação, **4**  
 Multipurpose Internet Mail Extensions (MIME), **733**  
 multitarefa, **8**  
 multithreading, 8  
 Musser, David, 892  
 mutable  
   demonstração, 971  
   especificador de classe de armazenamento, **200**  
   membro de dados, **971**, 971  
   palavra-chave, **971**

## N

naipes das cartas, representação dos, 336  
 name, atributo do elemento `input`, **1084**  
 name, função da classe `type_info`, **585**  
 namespace, 967  
   aninhado, **966**  
   apelido, 968  
   global, **966**  
   não identificado, **966**  
   predefinido, 966  
   qualificador de, 968  
 namespace, membro de, **965**  
 namespace, palavra-chave, **965**, 965, 966, 968  
 namespaces, escopo de, **202**  
 não fatal  
   erro de lógica, **38**  
   erro de tempo de execução, **10**  
   erro, 663  
 NÃO lógico (!), operador, 161, **162**, 162, 179, 969  
 não modificável, **292**  
 não-const, função-membro, 413  
 não-static, função-membro, 468  
 navegabilidade bidirecional na UML, **400**  
 navegador baseado em texto, **1072**  
 navegador Web, 753  
 navegador, 752, 753  
 NCSA (National Center for Supercomputing Applications), **732**  
 NDEBUG (pré-processor), 1026  
 negação lógica, 161, **162**  
 negate, objeto de função, 953  
 .NET Framework Class Library da Microsoft, 8  
 .NET, plataforma, **8**  
 Netscape Communicator, 1064  
 new, handler de, **658**  
 new, operador, **430**, 466, 467, 1011  
   chama o construtor-padrão, 435  
   falhas, 654, 663  
   função `new_handler`, 663  
   lançando `bad_alloc` em caso de falha, 657  
   retornando 0 em caso de falha, 656, 658  
 new, tratamento de falhas, 656  
 <new.h>, arquivo de cabeçalho, 656  
 new\_handler, função, 663  
 <new>, arquivo de cabeçalho, 656  
 nível de recuo, 99  
 nível de título (XHTML), **1066**  
 nível mais alto de precedência, 36  
 nó de rede, 818  
 nó mais à direita de uma subárvore, 834  
 nó substituto, 834  
 nó, **800**  
 noboolalpha, manipulador de fluxo, **632**  
 nó-filho, 834  
 nó-folha, **820**, 825, 834  
 Nome completo de computador:, campo, 731  
 nome de arquivo, **672**, 674  
 nome de classe definida pelo usuário, 58  
 Nome de computador, campo, 731  
 nome de domínio completamente qualificado (*fully qualified domain name* — FQDN), **730**  
 nome de domínio, **730**  
 nome de função desfigurado, 217  
 nome de papel na UML, **84**, 84  
 nome de tipo (enumerações), **197**  
 nome de um arquivo-fonte, 1026  
 nome de um array, **252**, 316  
 nome de um atributo, **1066**  
 nome de uma classe definida pelo usuário, 58  
 nome de uma variável de controle, **141**  
 nome de uma variável, **34**, 200  
 nome de variável global, 1007  
 nome de variável, 34, 835  
   de argumento, 61  
   de parâmetro, 61  
 nome qualificado, 530  
 nome subscrito de um elemento de array, 267  
 nome subscrito utilizado como um *rvalue*, 465  
 nome, conflito de atribuição de, 965, 966  
 nomes ocultos em escopos externos, 202  
 nomes significativos, 62  
 nó-pai, **821**, 834  
 nó-raiz da subárvore direita, 820  
 nó-raiz da subárvore esquerda, 820  
 nó-raiz, **820**, 825  
 noshowbase, manipulador de fluxo, 626, **630**  
 noshowpoint, manipulador de fluxo, **626**  
 noshowpos, manipulador de fluxo, 365, 626, **626**  
 noskipws, manipulador de fluxo, 626

not, palavra-chave de operador, **969**  
not\_eq, palavra-chave de operador, **969**  
not\_equal\_to, objeto de função, 953  
nota, **96**, 96  
notação científica, **116**, 615, 631  
notação de ponteiro, 333  
notação de ponteiro/deslocamento, 332  
notação de ponteiro/subscrito, 333  
notação fixa, 615, 626, 631  
notação infixa, **831**  
notação O, 276, **782**, 785, 787, 788, 793, 794  
notação pós-fixa, **831**  
notação posicional, 992  
nothrow, objeto, **658**  
nothrow\_t, tipo, 658  
nouppercase, manipulador de fluxo, 626, **631**  
nova linha ('\n'), sequência de escape de, **29**, 29, 33, 40, 346, 616, 866  
novo bloco de memória, 1012  
novos manipuladores de fluxo, 624  
nth\_element, 949  
NULL, **313**  
<numeric>, arquivo de cabeçalho, **901**, 934  
número aleatório, 195  
número binário, 864  
número correto de argumentos, 188  
número de argumentos, 188  
número de elementos em um array, 328  
número de linha, 835, 838, 839, 1026  
número de ponto flutuante de dupla precisão, **115**  
número de ponto flutuante de precisão simples, **115**  
número de ponto flutuante no formato científico, 631  
número de posição, **251**  
número de sinal, 1009  
número decimal, 179, 630, 864  
número ímpar, 179  
número inteiro, 31  
número não especificado de argumentos, 1003  
número perfeito  
    exercício, 244  
número real, 111  
números complexos, 408, 494  
números de ponto flutuante no formato científico, 631  
números mágicos, **258**  
números perfeitos, 244  
números primos, 951  
    exercício, 244  
números pseudo-aleatórios, **195**

## O

O(1), tempo, **782**  
O(log n), tempo, **785**  
O(n log n), tempo, **791**  
O(n), tempo, **782**  
O(n²), tempo, **782**  
Object Management Group (OMG), **19**  
ObjectSpace STL Tool Kit, exemplos do, 956, 1062  
objeto automático local, 392, 394  
objeto automático, 654, 663  
objeto de exceção, **650**, 650  
objeto de fluxo de entrada-padrão (cin), 33, 614, 671  
objeto de fluxo de saída padrão (cout), **28**, 614, 671, 1002  
objeto de função comparador, **913**, 915  
    less, **913**, 915  
objeto de função, 913, 915, **953**  
    binária, **953**, 954  
    predefinido da STL, 953  
objeto de uma classe derivada é instanciado, 532  
objeto de uma classe derivada, 548, 552, 553  
objeto deixa o escopo, 391  
objeto fila, 833  
objeto fluxo de entrada (cin), 31, **33**  
objeto grande, 210  
objeto host, **421**  
objeto local automático, 392  
objeto não identificado (não nomeado), 1014  
objeto sai de escopo, 659  
objeto String temporário, 477  
objeto temporário, 468  
objeto, 2, 3, 6, 8, **17**, 17, 57, 294  
objeto, orientação a, 16, 17  
objetos contêm apenas dados, 382  
objetos de função predefinidos, 953  
objetos de função  
    divides, 953  
    equal\_to, 953  
    greater, 953  
    greater\_equal, 953  
    less, 953  
    less\_equal, 953  
    less< int >, **913**, 913  
    less< T >, 923  
    logical\_and, 953  
    logical\_not, 953  
    logical\_or, 953  
    minus, 953  
    modulus, 953  
    multiplies, 953  
    negate, 953  
    not\_equal\_to, 953  
    plus, 953  
Observações de engenharia de software, **6**  
obtem valor de, 38  
obter (get) um valor, 66  
oct, manipulador de fluxo, 621, **621**, 626, 630  
octal (base 8), sistema de numeração, 621, 626, 871, 992  
octal, número, 179, 615, 630, 864, 871  
ocultação de dados, **65**, 66  
ocultação, 426  
ocultamento de informações, **17**, 325, 440  
ocultar dados private de clientes, 384  
ocultar uma representação interna dos dados, 442  
off-by-one, erro, **108**, 143, 144, 252  
offset, **581**  
ofstream, 671, 672, 673, 674, 675, 682, 688, 1005  
    open, função, **673**  
ofstream, construtor, 673  
Oito Rainhas, 308  
    abordagens de força bruta, 308  
Oito Rainhas, exercício de recursão, 228, 310

om (domínio de nível mais alto), 731  
OMG (Object Management Group), 19  
OOAD (análise e projeto orientados a objetos), 18  
OOD (projeto orientado a objetos), **17**, 17, 41, 45, 82, 123, 126, 228  
OOP (programação orientada a objetos), 2, 3, 6, 11, **17**, 18, 41, 502  
opção de compilador, 11  
open, função de ofstream, **673**  
operação (UML), 59  
operação abstrata na UML, **586**  
operação de fluxo falhou, 634  
operação na UML, **17**, 59, 83, 228, 229, 231, 401, 404, 587  
operações de inserção de fluxo de concatenação, **34**  
operações de inserção de fluxo de encadeamento, **34**  
operações de inserção de fluxo em cascata, **34**  
operações que podem ser realizadas nos dados, **441**  
operador \* sobrecarregado, 659  
operador +=, 706  
operador -> sobrecarregado, 659  
operador aritmético binário, 116  
operador aritmético, **35**, 35  
operador binário, **33**, 35, 162  
operador comutativo, 454  
operador condicional (:?), **100**  
operador condicional ternário (:?:), **100**, 225, 452  
operador de adição unário (+), 116  
operador de atribuição de adição (+=), **120**, 121  
operador de atribuição, **120**, 451  
    %=, operador de atribuição de módulo, 121  
    \*=:, operador de atribuição de multiplicação, 121  
    /=, operador de atribuição de divisão, 121  
    +=, operador atribuição de adição, 120, 121  
    =, **33**, 40, 41, 397, 452  
    -=, operador de atribuição de subtração, 121  
    função operadora sobrecarregada, 467  
operador de bits de deslocamento para a direita (>>), 451  
operador de bits de deslocamento para a esquerda (<<), 451, 857  
operador de coerção unário, **115**  
operador de coerção, **111**, 115, 116, 190, 331, **468**, 487  
operador de conversão, **468**  
operador de decremento unário (--), 121  
operador de decremento  
    --, **121**, 121  
    sobrecarregando, 479  
operador de desreferência (\*), **314**, 315  
operador de endereço (&), 313, 315, 316, 325, 452, 972  
operador de igualdade (==), **38**, 38, 39, 458, 895, 896  
operador de incremento unário (++), 121  
operador de ponteiro para membro de classe  
    .\*, **972**  
    ->\*, **972**

operador de resolução de escopo binário (: :), **76**, 436, 965, 972, 976

operador de seleção de membro ponto (.), 382, 430, 558, 659

operador de subscrito [], 705

operador de subscrito de map, 918

operador de subscrito, [], utilizado com strings, 705

operador de subtração unário (-), 116

operador ponto (.), **59**, 382, 430, 558, 659

operador relacional, **38**, 38, 39, 144

Operador sizeof quando aplicado a um nome de array retorna o número de bytes no array, 328

operador unário, **116**, 162, 313

operador vírgula (,), **144**, 225, 452

operador

- ! (operador NÃO lógico), 161, **162**
- != (operador de desigualdade), 38
- % (operador módulo), 35
- %=, de atribuição de módulo, 121
- && (operador E lógico), 161, **161**
- \* (operador de desreferência ou de indireção), **314**, 315
- \* (operador de multiplicação), 35
- \*=, de atribuição de multiplicação, 121
- / (operador de divisão), 35
- /=, de atribuição de divisão, 121
- || (operador OU lógico), 161, **161**
- + (operador de adição), 33, 35
- +=, de atribuição de adição, 120
- < (operador menor que), 38
- << (operador de inserção de fluxo), **28**, **29**, 33, 34
- <= (operador menor que ou igual a), 38
- = (operador de atribuição), 33, 162
- =, de atribuição de subtração, 121
- == (operador de igualdade), 38, 162
- > (operador maior que), 38
- >= (operador maior que ou igual a), 38
- >> (operador de extração de fluxo), 33, 34
- condicional (: :), **100**
- const\_cast, **964**
- de atribuição aritmético, 120, 121
- de atribuição de adição (+), **120**, 121
- de atribuição, **120**
- de decremento (--), **121**, 121
- de decremento prefixado, **121**, 121
- de endereço (&), **313**, 315
- de incremento (++), **121**, 121
- de incremento prefixado, **121**, 121, 122, 123
- de pós-decremento, **121**, 121
- de pós-incremento, **121**, 121, 123
- de resolução de escopo binário (: :), **76**
- de seleção de membro ponto (.), 382
- de seleção de membro seta (->), 382
- delete, **430**, 467, 585, 659, 800, 1011
- dynamic\_cast, **582**, 661
- multiplicativo (\*, /, %), **116**
- new, **430**, 466, 467, 1011
- ponto (.), **59**
- sizeof, **327**, 329, 382, 429, 685, 687, 701, 800, 848, 873, 1024
- ternário, **100**
- unário de resolução de escopo (: :), **215**

- unário mais (+), 116
- unário menos (-), 116

operador, palavras-chave de, **968**, 969

- demonstrando as, 969

operadores . \* e ->\*, 972, 973

operadores de atribuição aritméticos, 120, 121

operadores de igualdade (== e !=), 98, 161

operadores lógicos, **161**

- palavras-chave, 969

operadores multiplicativos (\*, /, %), **116**

operadores que podem ser sobrecarregados, 452, 453

operadores relacionais >, <, >= e <=, 161

operadores, associatividade de, 164

operando direito, 29

operando, **29**, 33, 35, 100, 362, 838, 839

operador void \*, função-membro, 636, 674, 675

operator!, função-membro, 457, 458, 636, 673

operator!=, 467, 468, 893, 894

operator() sobrecarregado, 953

operator(), 494

operator, palavra-chave, 452

operator[]

- versão const de, 468

operator+, 452

operator++( int ), 479

operator++, 479, 484

operator<, 478, 893, 894

operator<<, 457, 465, 469, 500

operator<=, 893, 894

operator=, 467, 894

operator==, 467, 468, 478, 893, 894

operator>, 893, 894

operator>=, 478, 893, 894

operator>>, 455, 457, 465, 469

option (XHTML), elemento, **1089**

options, tipo de solicitação HTTP, 729

OR lógico de bitwise (||), operador, 950

or, palavra-chave de operador, **969**

or\_eq, palavra-chave de operador, **969**

ordem l (notação O), 782

ordem correta dos argumentos, 188

ordem de avaliação, 225

- dos operadores, 36, 53

ordem de classificação, 936, 938

ordem do quadrado de n (notação O), **782**

ordem em que as ações executam, **94**, 104

ordem em que construtores e destrutores são chamados, 393

ordem em que os destrutores são chamados, 392

ordem em que os operadores são aplicados aos seus operandos, 225

ordem log n (notação O), **785**

ordem n (notação O), **782**

org (domínio de nível mais alto), 731

orientada para a ação, **17**

ostream, 614, 675, 681, 688, 695

ostream, classe

- função seekp, **675**
- função tellp, **677**

ostream\_iterator, **896**, 896

ostreamstring, **719**, 719, 720

otimização de compilador, 1009

otimização, 1009

otimizações em constantes, 412

otimizando o compilador de Simple, 843

OU exclusivo sobre bits (^), operador, **853**, 853, 855, 857, 858, 969

OU exclusivo sobre bits (^=), operador de atribuição, **860**

OU inclusivo sobre bits (|), operador, **853**, 853, 855, 858, 969

OU inclusivo sobre bits (|=), operador de atribuição, **860**

OU lógico (| |), operador, 161, **161**, 179, 857, 969

out\_of\_bounds, exceção, 907

out\_of\_range, exceção, **662**, 705, 706, 908, 950

outros conjuntos de caracteres, 704

oval, 53

overflow, 650

overflow\_error, exceção, **662**

overhead de função virtual, 924

overhead de função, 1024

overhead de tempo de execução, 579

overhead de uma chamada de função extra, 468

overhead de uma chamada de função, 1024

## P

$\pi$ , 179

p (parágrafo) (XHTML), elemento, 1066

</p> (tag final de parágrafo XHTML), **1066**

<p> (tag inicial de parágrafo XHTML), **1066**

Package, hierarquia de herança, 594

pacote de imagens gráficas, 594

pacote, 818, **1057**

padrão de 1s e 0s, 669

Paint Shop Pro, 1070

pair, 915, 926

palavra, 362, 848

palavra-chave, **28**, 97, 98

- and, **969**
- and\_eq, **969**
- auto, 200
- bitand, **969**
- bitor, **969**
- catch, **649**
- class, 219, 597
- compl, **969**
- const na lista de parâmetros de função, 209
- enum, **197**, 199
- explicit, **488**
- extern, 201
- inline, **208**, 379
- mutable, **971**
- namespace, **965**, 965, 966, 968
- not, **969**
- not\_eq, **969**
- or, **969**
- or\_eq, **969**
- private, **64**, 65
- public, **58**, 64, 65
- static, 201
- tabela de palavras-chave, 97
- template, **597**
- throw, **650**
- try, **647**



- typedef, **614**, 704, 719, **849**, 894, 913, 915, 918
- typename, 219, **597**, 597
- void, **59**
- xor, **969**
- xor\_eq, **969**
- palavras e frases descritivas (estudo de caso OOD/UML), 124, 125
- palíndrome, função, 961
- palíndromo, 831
- papel na UML, 84
- par de chaves {}, 40, 81
- par mais interno de parênteses, 36
- par nome-valor, 753
- parágrafo, tag de, **1066**
- paralelogramo, 503
- parâmetro de exceção, **649**
- parâmetro de operação na UML, 229, 231, 232
- parâmetro de referência constante, 210
- parâmetro de referência, 209, **210**, 210
- parâmetro de template de tipo, **597**
- parâmetro de template não-tipo, **605**, 605
- parâmetro de template, **597**, 603
- parâmetro de tipo formal, **219**, 219
- parâmetro de tipo, 219, 597, 599, 600, 605
- parâmetro formal, **188**
- parâmetro na UML, 62, 229, 231, 232
- parâmetro, **60**, 60, 61, 62, 200
- parênteses (()), 36, 115, 116
- parênteses aninhados, **36**
- parênteses embutidos, **36**
- parênteses para forçar a ordem de avaliação, 41
- parênteses redundantes, **37**, 161
- parênteses vazios, 59, 61
- pares de iteradores, 899
- parte fracionária, 115
- parte inferior de uma pilha, 813
- parte superior de uma pilha, **799**, 813
- partes intercambiáveis padronizadas, 18
- partial\_sort, algoritmo STL, 949
- partial\_sort\_copy, algoritmo STL, 949
- partial\_sum, algoritmo STL, 901
- particionamento, passo de, 368
- partition, algoritmo STL, 901
- Pascal, Blaise, 7
- passagem da esquerda para a direita por uma expressão, 831
- passagem de mensagem na UML, 295
- passagem por referência, **209**, 210, 267, 268, 312, 316, 317, 319, 324
  - com argumentos de ponteiro, 210, **315**
  - com parâmetros de referência, 210, **315**
  - com um argumento de ponteiro utilizado para elevar ao cubo o valor de uma variável, 317
- passagem por valor, **209**, 210, 267, 315, 316, 318, 325
  - utilizada para elevar o valor de uma variável ao cubo, 316
- passando argumentos por valor e por referência, 210, 211
- passando arrays e elementos de array individuais a funções, 268
- passando arrays para funções, 267
- passar o tamanho de um array como um argumento, 267
- passar o tamanho de um array, 327
- passar objetos grandes, 210
- passar opções para um programa, 336, 1005
- passar um array inteiro, 268
- passar um elemento do array, 268
- passar um nome de arquivo para um programa, 1005
- passar um objeto por valor, 398, 399
- passar uma variável de estrutura para uma função, 849
- Passeio do Cavalo, 306
  - abordagens de força bruta, 308
  - teste do passeio fechado, 308
- passo de partição em quicksort, 797
- passo de recursão, **221**, 222, 224, 797
- passo recursivo, 368
- password (type), valor do atributo, 742
- path, atributo em cookies, 753
- pedido de terminação proveniente do sistema operacional, **1009**
- peek, função de istream, **619**
- percorrendo um labirinto, exercício de recursão, 228
- percorrer a subárvore direita, 827
- percorrer a subárvore esquerda, 827
- percorrer para frente e para trás, 812
- percorrer para trás, 718
- percorrer um contêiner, 799
- percorrer uma árvore binária, 821, 828
- percorrer uma lista, 812
- percorrer, 718
- percurso na ordem de nível de uma árvore binária, 828, 834, 835
- percurso na ordem de uma árvore binária, exemplo de recursão, 228
- percurso na ordem, **821**, 835
- percurso na pós-ordem de uma árvore binária, exemplo de recursão, 228
- percurso na pós-ordem, **821**, 833, 834
- percurso na pré-ordem de uma árvore binária, exemplo de recursão, 228
- percurso na pré-ordem, **821**
- perda de dados, 634
- permutação lexicográfica, 949
- permutação, 949
- permutar valores (em algoritmos de classificação), 787, 788
- persistência de dados, **669**
- pesquisa binária recursiva, 228, 797
- pesquisa binária, 276
  - algoritmo de, **783**, 785
- pesquisa de arrays, **276**
- pesquisa de blocos de memória, 874
- pesquisa linear de um array, 277
- pesquisa linear recursiva, 228, 310, 797
- pesquisa linear, **276**
- pesquisa linear, algoritmo de, 782, 785
- pesquisa, 442, 799, 934, 935
- pesquisar dados, **781**
- pesquisar em uma lista vinculada, 802, 845
- pesquisar recursivamente uma lista, 833
- pesquisar strings, 347, 868
- pesquisar uma lista vinculada, exercício de recursão, 228
- Peter Minuit, problema de, 149, 180
- PhoneNumber, classe, 500
- PhotoShop Elements, 1070
- pi, 53
- PI, 1023
- picture.html, 1071
- pilha de chamada, 322
- pilha de classe float, 596
- pilha de classe int, 596
- pilha de classe string, 596
- pilha de execução do programa, **204**
- pilha, **204**, 442, 599, 600, **799**, 800, 801, 812, 815
  - desempilhamento de, **649**, 654, 655, 663
- pilhas implementadas com arrays, 440
- pilhas utilizadas por compiladores, 831
- pipe (), **1002**
- pipe, **735**
- piping, **1002**
- pixel, **1072**
- plataforma de hardware, 6
- plataforma, 11
- Plauger, P. J., 6
- plus, objeto de função, 953
- pointer (STL), 895
- polimorfismo, 158, 541, **546**, 546, 547, 548, 560, 843, 924
  - como uma alternativa à lógica switch, 594
  - e referências, 579
- polinômio de segundo grau, 37
- polinômio, 37
- Polynomial, classe, 500
- ponteiro & e \*, operadores de, 315
- ponteiro constante, 332, 430
  - para dados constantes, 320, 324, 325
  - para dados não constantes, 320, **323**, 323
  - para um inteiro constante, 324
- ponteiro de cauda, 845
- ponteiro de função, 341, 579, 580, 581, 953
- ponteiro de posição do arquivo, **675**, 685, 695
- ponteiro implícito, 426
- ponteiro não constante para dados constantes, 320, **321**, 322
- ponteiro não constante para dados não constantes, 320, **320**
- ponteiro nulo (0), **313**, 314, 674, 800, 801, 834, 1012
- ponteiro oscilante, **466**, 477
- ponteiro para frente, 812, 813
- ponteiro para trás, 812, 813
- ponteiro para um objeto, 322, 379
- ponteiro para uma função, 341, 1007
- ponteiro para void (void \*), 331, 332
- Ponteiro this acessando implícita e explicitamente os membros de um objeto, 431
- ponteiro viable, 581, 582
- ponteiro, 312, 330
- ponteiros declarados como const, 323, 324
- ponteiros e arrays, 332
- ponteiros e subscrito de array, 332, 333
- ponteiros para armazenamento dinamicamente alocado, 430, 467

- ponto de entrada, 165
- ponto de interrupção, **1098**, **1112**
  - círculo sólido marrom, 1100
  - desativar, **1102**
  - inserir, 1100, 1103, 1114, 1118
  - seta amarela no modo de interrupção, **1101**
- ponto de lançamento, **649**
- ponto de saída de uma instrução de controle, 165
- ponto de venda, sistema de, 680
- ponto decimal, 111, 116, 149, 615, 626
- ponto flutuante, 621, 622, 626, 631
- ponto flutuante, divisão, 115
- ponto flutuante, exceção de, **1009**
- ponto flutuante, limites de tamanho de, 192
- ponto flutuante, número de, **111**, 116
  - de dupla precisão, **115**
  - de precisão simples, **115**
  - tipo de dados double, **111**
  - tipo de dados float, **111**
- ponto flutuante, valores de, exibidos nos
  - formatos-padrão, científico e fixo, 631
- ponto forçado de fração decimal, 615
- ponto-e-vírgula (;), **28**, 40, 62, 102, 1022
- ponto-e-vírgula que termina uma definição de
  - estrutura, 847
- pop, função de adaptadores de contêiner, **919**
- pop, função-membro de priority\_queue, **923**
- pop, função-membro de queue, **921**
- pop, função-membro de stack, **919**
- pop\_back, função-membro de list, **1136**
- pop\_front, 908, 912, 921
- pop\_heap, algoritmo STL, **948**
- pôquer, programa de jogo de, 360
- porcentagem (%) (operador módulo), sinal de, **35**
- portabilidade, 11
- portal.cgi, 748
- portal.cpp, 750
- portável, 6
- pós-decrementar, **121**, 123
- pós-decremento, operador de, **121**, 121
- posição atual de um fluxo, 675
- posição dos 2s, 992
- posição na memória, **34**
- posição temporária, 842
- pós-incrementar um iterador, 900
- pós-incrementar, **121**, 484
- pós-incremento, operador de, **121**, 121, 123
- post, tipo de solicitação HTTP, **728**, 745
- post, tipo de solicitação, 1084
- post.cpp, 746
- postOrderTraversal, 827
- pow, função, 37, **147**, 147, 184
- power, 129
- #pragma, diretiva de pré-processador, **1025**
- precedência de operadores, 35, 36, 123, 164, 859, 860
- precedência, 35, 36, 37, 40, 41, 123, 144, 161, 162, 225
  - do operador condicional, 100
  - gráfico de, 41
  - não alterada pela sobrecarga, 452
- precisão de números de ponto flutuante, **621**
- precisão de um valor de ponto flutuante, **111**
- Precisão de valores de ponto flutuante, 622
- precisão, **116**, 615, 620
  - formatação de um número de ponto flutuante, 116
- precisão-padrão, **116**
- precision, função de ios\_base, **621**
- pré-decrementar, **121**
- preencher com caracteres especificados, 615
- preenchimento de caracteres, 620
- preenchimento em uma estrutura, 862
- preenchimento, **862**
- pré-incrementar, **121**, 481, 484
- pré-processador, 10, 189, **1022**
- pré-processamento, 9
- primeira passagem do compilador de Simple, 838, 839, 840, 843
- primeiro a entrar, primeiro a sair (*first-in, first-out* – FIFO), **442**, 442, 817, 893, 911, 921
- primeiro refinamento, **109**, 117, 337
- primos, 244
- principal, 147
- princípio do menor privilégio, **200**, 261, 270, 318, 319, 327, 335, 384, 412, 675, 899, 1006, 1007
- print, comando de depurador, **1115**
- printArray, template de função, 597
- priority\_queue, classe adaptadora, **922**, 923
  - função empty, **923**
  - função pop, **923**
  - função push, **923**
  - função size, **923**
  - função top, **923**
- private static, membro de dados, 436
- private
  - classe básica, 540
  - dados, da classe básica não podem ser acessados a partir da classe derivada, 516
  - especificador de acesso, **64**, 64, 65, 399, 400
  - herança, 502, **504**, 539, 814
  - herança, como uma alternativa à composição, 539
  - membro de dados, 395, 396
  - membros, de uma classe básica, 504
- privilégios de acesso, 320, 324
- probabilidade, **192**
- procedimento, **94**, **183**
- procedure pura, **382**
- processador de texto, 345
- processamento de acesso instantâneo, 689
- processamento de arquivo de dados brutos, 669
- processamento de arquivo de dados formatados, 669
- processamento de arquivo, 613, 615
- processamento de dados comerciais, 699
- processamento de fluxo de string, **719**
- processamento de listas, 802
- processamento de textos, 372
- processamento de transações, 915
- processamento em lotes, **4**
- processamento polimórfico de exceção, 654
- processo de compilação para um programa Simple, 841
- processo de refinamento, 109
- processo do projeto, 18, 41, 45, 229, 232
- produto de inteiros ímpares, 178
- programa de adição que exibe a soma de dois números, 31
- programa de arquivo de múltiplas fontes, 1005
- Programa de classificação para múltiplos propósitos utilizando ponteiros de função, 342
- programa de computador, **3**
- programa de consulta de crédito, 677
- programa de conta bancária, 689
- programa de impressão de texto, 27
- programa de múltiplos arquivos de código-fonte
  - processo de compilação e vinculação, 77
- programa de processamento de crédito, 682
- programa de processamento de fila, 819
- programa de processamento de transação, 689
- programa de rolagem de dados utilizando um array em vez de switch, 260
- Programa de teste da classe BasePlusCommissionEmployee, 513
- programa driver, **71**
- programa gerenciador de tela, 547
- Programa polimórfico de operações bancárias usando a hierarquia Account, exercício de, 594
- programa processador de arquivo, 699
- programa simples de pilha, 815
- programa tradutor, **5**, **5**
- programa, 3
- programação de linguagem de máquina, 362
- programação defensiva, 144
- programação estruturada, 2, 3, 7, 8, 94, **95**, 160, 165, 1012
  - resumo, 165
- programação genérica, **596**, 891, 892, 895, 956, 1062
- programação orientada a objetos (*object-oriented programming* – OOP), 2, 3, 6, 11, **17**, 18, 41, 377, 502
- programação polimórfica, 563, 564, 581, 582
- programação sem goto, 95
- programador de código-cliente, 77, 78
- programador de computador, **3**
- programador de implementação de classe, 77, 78
- programador, **3**
- programar no específico, 546
- programar no geral, 546, 594
- programas tolerantes a falhas, **645**
- projeto orientado a objetos (*object-oriented design* – OOD), **17**, 17, 41, 45, 82, 123, 126, 139, 228, 399, 427
- projetos, 1007
- promoção de inteiro, **115**
- promoção, **115**, 115
- prompt de linha de comando, 1002
- prompt de shell no Linux, 11
- prompt, **33**, 113, 1002
- propósito do programa, 28
- Propriedades de sistema, janela, 731
- protected, 504, 505
- protected, classe básica, 540
- protected, especificador de acesso, 377
- protected, herança, 502, **504**, 539
- protocolo, **1084**
- protótipo de função
  - escopo de, **202**, 202
  - nomes de parâmetro opcionais, 74

ponto-e-vírgula no final de, 74  
 provedores de serviço da Internet (*Internet service provider* – ISP), 1084  
 pseudocódigo, 18, 94, 94, 95, 98, 99, 104, 116, 117, 338  
   do topo, 109  
   dois níveis de refinamento, 111  
   primeiro refinamento, 109  
   refinamento passo a passo de cima para baixo, 109  
   segundo refinamento, 109  
 public static, função-membro, 436  
 public static, membro de classe, 436  
 public  
   classe básica, 539  
   especificador de acesso, 58, 64, 65, 399, 400  
   herança, 502, 504  
   interface, 65  
   membro, de uma classe derivada, 504  
   palavra-chave, 58, 64, 404  
   serviços, de uma classe, 74  
 pular caracteres de espaço em branco, 626  
 pular espaço em branco, 620  
 pular o código restante no loop, 160  
 pular o restante de uma instrução switch, 159  
 push, 602, 813, 814  
 push, função de adaptadores de contêiner, 919  
 push, função-membro de priority\_queue, 923  
 push, função-membro de queue, 921  
 push, função-membro de stack, 919  
 push\_back, função-membro de vector, 903  
 push\_front, função-membro de deque, 911  
 push\_front, função-membro de list, 908  
 push\_heap, 946, 948  
 put, função-membro, 615, 616, 617  
 put, ponteiro de posição de arquivo, 681, 685  
 put, ponteiro, 675  
 put, tipo de solicitação HTTP, 729  
 putback, função de istream, 619

## Q

quadrado dos lados de um triângulo, 179  
 quadrado, 137  
 quadrados de vários inteiros, 836  
 Quadralay Corporation's, site Web, 1060  
 quadro de pilha, 205  
 qualificador de tipo, 270  
 qualityPoints, 244  
 quantias monetárias, 149  
 quantidades escalares, 267  
 quebra de linha, 1074  
 QUERY\_STRING, variável de ambiente, 737, 742, 745  
 querystring.cpp, 740  
 questões de segurança relacionadas com o Common Gateway Interface, 775  
 queue, classe adaptadora, 921, 922  
   função back, 921  
   função empty, 921  
   função front, 921  
   função pop, 921  
   função push, 921  
   função size, 921  
 queue, classe, 442

Queue, definição do template de classe, 818  
 <queue>, arquivo de cabeçalho, 191, 895, 922, 923  
 Quick Info, caixa, 1102  
 quicksort, algoritmo, 368, 797  
 quicksort, exercício de recursão, 228  
 quit, comando de depurador, 1117

## R

radianos, 184  
 radio (XHTML), valor de atributo, 742, 1086  
 raio de um círculo, 138  
 raise, função, 1009, 1009  
 raiz quadrada, 184, 622  
 rand, função, 192, 303  
 rand, semear a função, 195  
 RAND\_MAX, constante simbólica, 192  
 random\_shuffle, algoritmo STL, 901, 932, 932  
 Rational Software Corporation, 19, 45  
 Rational Unified Process™ (RUP), 45  
 Rational, classe, exercício, 409  
 RationalNumber, classe, exercício, 500  
 rbegin, função-membro da classe string, 718  
 rbegin, função-membro de contêineres, 894  
 rbegin, função-membro de vector, 905  
 RDBMS (sistema de gerenciamento de banco de dados relacional — *relational database management system*), 730  
 rdstate, função de ios\_base, 635  
 read, função de istream, 620, 620, 681, 687, 695  
 readcookie.cpp, 756  
 realizar uma ação, 28  
 realizar uma tarefa, 59  
 realloc, 1011, 1012  
 reatribuir uma referência, 213  
 receber o valor de, 38  
 Rectangle, 409  
 Rectangle, classe, exercício, 409  
 recuo, 40, 98, 99, 101, 142  
 recuperação de erros, 634  
 recursão infinita, 466  
 recursão, 221, 221, 226, 246  
 recursão, exercícios de  
   pesquisa binária, 797  
   pesquisa linear, 797  
 recursos de carreira, 1060  
 recursos STL na Internet, 955  
 rede de computadores, 4, 817, 818  
 rede local (local area network – LAN), 4  
 'rede peão', 4  
 Rede, diálogo, 731  
 redirecionar entrada/saída em sistemas UNIX, Linux, Mac OS X e Windows, 1002  
 redirecionar entradas de modo que provenham de um arquivo, 1002  
 redirecionar saída de um programa para entrada de outro programa, 1002  
 redirecionar saídas para um arquivo, 1002  
 referência a um membro de dados private, 395  
 referência a um objeto, 379  
 referência a uma constante, 211, 212  
 referência a uma variável automática, 212, 213  
 referência antecipada, 838, 839

referência constante, 467  
 referência de entidade (XHTML), 1074  
 referência de entidade de caractere, xxxiv  
 referência não inicializada produz um erro de sintaxe, 213  
 referência oscilante, 212  
 referência para dados constantes, 322, 323  
 referência para um int, 210  
 referência, 312, 612, 895  
 referenciando elementos do array com o nome do array e com ponteiros, 333  
 referenciar elementos do array, 333  
 referenciar um valor diretamente, 312  
 referenciar um valor indiretamente, 312  
 referências devem ser inicializadas, 212  
 referências não-resolvidas, 839, 1005  
 refinamento passo a passo de cima para baixo, 109, 111, 337  
 refinamento passo a passo, 337  
 Refresh, cabeçalho, 745  
 register, declaração, 201  
 register, especificador de classe de armazenamento, 200  
 registrar uma função com atexit, 1007  
 registro de ativação, 205, 206  
 registro de transação, 700  
 registro, 669, 671, 688, 700  
 regra de empilhamento, 165  
 regra de negócio, 730  
 regra geral, 160  
 regras de precedência de operador, 35  
 regras de promoção, 189  
 regras para formar programas estruturados, 165  
 reinterpret\_cast, operador, 681, 681, 682, 687  
 reinventar a roda, 6  
 relação áurea, 223  
 relacionamento de muitos para um, 85  
 relacionamento de um para muitos, 85, 915  
 relacionamento de um para um, 85  
 relacionamento do todo/parte, 84  
 relacionamento é um (herança), 502, 540, 974, 976  
 relacionamento hierárquico entre a função-chefe e as funções-trabalhador, 183  
 relacionamento tem um (composição), 84, 421, 502  
 relançar uma exceção, 651  
 rem, instrução Simple, 835, 836, 838  
 remove, algoritmo STL, 901, 927, 927, 928  
 remove, função-membro de list, 911  
 remove\_copy, algoritmo STL, 901, 927, 928, 928  
 remove\_copy\_if, algoritmo STL, 901, 927, 930  
 remove\_if, algoritmo STL, 901, 927, 928, 928  
 rend, função-membro da classe string, 718  
 rend, função-membro de contêineres, 894  
 rend, função-membro de vector, 905  
 repetição controlada por contador, 104, 104, 105, 106, 107, 108, 114, 117, 141, 142, 226  
   com a instrução for, 143  
 repetição controlada por sentinela, 109, 110, 111, 113, 114  
 repetição definida, 104



- repetição indefinida, **109**
  - repetição, 167, 169
  - repetição, estrutura de, **96**
  - repetição, instrução de, 98, **103**, 110, 836
    - do...while, 150, 151, 169
    - for, **143**, 143, 144, 169
    - while, 103, 114, 141, 142, 150, 169
  - repetição, término da, 103
  - repetitividade da função rand, 195
  - replace, algoritmo STL, 901, 930, **930**
  - replace, função-membro da classe string, **714**, 714
  - replace\_copy, algoritmo STL, 901, 930, **930**
  - replace\_copy\_if, algoritmo STL, 901, 930, **932**
  - replace\_if, algoritmo STL, 901, 930, **930**
  - representação de dados, **441**
  - representação gráfica de uma árvore binária, 821
  - representação interna de uma string, 469
  - representação numérica do caractere, 154
  - requisitos de sistema, **44**
  - requisitos, 11, **18**, 44
  - reset (type), valor do atributo, 742
  - reset de bitset, **951**
  - resize, função-membro da classe string, **712**
  - resolução de escopo (: :), operador unário de, **215**, 215
  - resolução de escopo binário (: :), operador de, 436, 600, 606, 965, 972, 976
  - restaurar o estado de um fluxo a um estado 'válido', 635
  - resto da divisão de inteiros, 35
  - resultados do exame, problema dos, 118, 119
  - retângulo arredondado (para representar um estado em um diagrama de estados UML), **169**
  - reticências (...) no protótipo de uma função, 1003
  - retirar de uma pilha, **204**
  - Retornando indicadores de erros das funções set da classe Time, exercício, 409
  - retornando uma referência a um membro de dados private, 395, 396
  - retornando uma referência de uma função, 212
  - retornar um resultado, 28, 188
  - retorno de carro ('\r'), 864, 866
  - return, instrução, 29, 188, 221, 844
  - Return, tecla, 33
  - reusabilidade, 327, 599, 601
  - reutilização de software, 6, 29, 183, 502, 596, 599, 601, 973
  - reutilização, 18, 70, 381
  - reutilizando componentes, 8
  - reverse, algoritmo STL, 901, 938, **940**
  - reverse\_copy, algoritmo STL, 901, 940, **940**
  - reverse\_iterator, **718**, 894, 895, 899, 905
  - rfind, função-membro da classe string, **713**
  - Richards, Martin, 6
  - right value (rvalue), 165
  - right, manipulador de fluxo, **149**, 626, **626**
  - Ritchie, D., 6
  - rodapé da tabela (tabela XHTML), **1081**
  - rodapé da tabela (XHTML), **1081**
  - Rogue Wave, 8
  - rotate, algoritmo STL, 901, 949
  - rotate\_copy, algoritmo STL, 901, 949
  - rótulo (XHTML), elemento, **1086**
  - rótulo especificado em uma instrução goto, **1012**
  - rótulo, **202**
  - rows, atributo do elemento textarea, **1086**
  - rowspan, atributo do elemento th, **1081**
  - RTTI (informações de tipo em tempo de execução — *run-time type information*), **547**, 582
  - Rumbaugh, James, 18, 19
  - run, comando de depurador, **1114**
  - runtime\_error, classe
    - função what, 650
  - runtime\_error, exceção, **646**, 654, 661, 662
  - rvalue ('right value'), **165**, 212, 465, 468
- ## S
- saída armazenada em buffer, **615**
  - saída de caracteres, 615, 616
  - saída de dados, 10
  - saída de fluxo, 614
  - saída de inteiros, 615
  - saída de itens de dados de tipos predefinidos, 615
  - saída de letras maiúsculas, 615
  - saída de tipos de dados-padrão, 615
  - saída de um valor de ponto flutuante, 626
  - saída de valores de ponto flutuante, 615
  - saída de variáveis char \*, 616
  - saída em tela, 843
  - saída não armazenada em buffer, **615**
  - saída não formatada, 615
  - saída para strings na memória, 191
  - saída para strings na memória, realizar, 719
  - saídas acumuladas, 33, 34
  - sair de uma estrutura profundamente aninhada, 1012
  - sair de uma função, **29**
  - SalariedEmployee, arquivo de cabeçalho da classe, 568
  - SalariedEmployee, arquivo de implementação da classe, 569
  - salário bruto, 135
  - SalesPerson, definição da classe, 384
  - SalesPerson, definições de função-membro da classe, 385
  - SavingsAccount, classe, 448
  - scientific, manipulador de fluxo, 626, **631**
  - Screen, classe (estudo de caso ATM), 83, 84, 85, 229, 292, 293, 294, 295, 296, 401
  - script CGI simples, 733
  - script CGI, **731**, 732, 734, 735, 748, 1084
  - script, **1065**
  - search, algoritmo STL, 901, 949
  - search\_n, algoritmo STL, 901, 949
  - seção 'administrativa' do computador, 3
  - seção 'receptora' do computador, 3
  - seção cabeçalho, 1065, 1066
  - seção corpo, 1065
  - seção de 'envio' do computador, 3
  - seção de 'produção' do computador, 3
  - seção de armazenamento do computador, 4
  - Seção especial: construindo seu próprio compilador, 835
  - Seção especial: construindo seu próprio computador, 362
  - second, membro de dados de pair, **915**
  - seek get, 675
  - seek put, 675
  - seekg, função de istream, **675**, 695
  - seekp, função de ostream, **675**, 685
  - segunda passagem do compilador Simple, 843
  - segundo refinamento, **109**, 110, 118, 337
  - segurança, 399, 755, 761
  - segurança, risco de, 755
  - seleção única, 167
  - seleção, 167, 169
  - seleção, estrutura de, **96**
  - seleção, instrução de, 98
  - selecionar uma substring, 478
  - select (form), elemento XHTML, 742
  - select (XHTML), elemento, **1089**
  - selected, atributo do elemento option, **1089**
  - semente, 195
  - semicontêiner, **892**, 893
  - seno trigonométrico, 184
  - seno, 184
  - separando a interface da implementação, **73**
  - seqüência de entrada, **896**
  - seqüência de escape, **29**, 30
    - \' (caractere de aspas simples), 29
    - \" (caractere de aspas duplas), 29
    - \\ (caractere de barras invertidas), 29
    - \a (alerta), 29
    - \n (nova linha), 29
    - \r (retorno de carro), 29
    - \t (tabulação), 29, 156
  - seqüência de inteiros, 177
  - seqüência de mensagens na UML, **294**
  - seqüência de números aleatórios, 195
  - seqüência de saída, **896**
  - seqüência, 166, 167, 169, **896**, 934, 936, 938, 940, 948, 949
  - seqüência, estrutura de, **96**
  - seqüência, instrução de, 98
  - seqüências de caractere, 680
  - Série de Fibonacci, exercício, 246
  - serviços de uma classe, 66
  - servidor de arquivos, **4**
  - servidor DNS (Domain Name System), **731**
  - servidor Web local, **730**
  - servidor Web remoto, **730**
  - servidor Web, **728**, 732, 752, **1064**, 1083, 1084
  - servidor, 761
  - set e get, funções, 66
  - set, comando de depurador, **1118**
  - set, contêiner associativo, 915
  - set, função, 66, 425
  - set, operações, da Standard Library, 942
  - set\_difference, algoritmo STL, 940, **940**
  - set\_intersection, algoritmo STL, 940, **940**
  - set\_new\_handler especificando a chamada de função quando new falha, 658
  - set\_new\_handler, função, **656**, 658
  - set\_symmetric\_difference, algoritmo STL, 940, **940**
  - set\_terminate, função, **653**
  - set\_unexpected, função, 653, 661

- set\_union, algoritmo STL, 940, **943**
- <set>, arquivo de cabeçalho, 191, 895, **913**, 915
- seta (->), operador de seleção de membro, 382, 430
- seta amarela no modo de interrupção, **1101**
- seta de navegabilidade na UML, **400**
- seta de transição, **96**, 96, 98, 99, 103, 104
- seta em um diagrama de seqüências na UML, 295
- seta, 53, 96
- setbase, manipulador de fluxo, **621**, 621
- Set-Cookie:, cabeçalho HTTP, **753**
- setfill, manipulador de fluxo, 365, **378**, 626, **628**, 628, 629
- setprecision, manipulador de fluxo, **116**, 149, 621, 622
- setw, manipulador de fluxo, **149**, 253, 347, 455, 623, 626
- Shakespeare, William, 371
- short int, tipo de dados, 158
- short, tipo de dados, 158, 189
- showbase, manipulador de fluxo, 626, **630**
- showpoint, manipulador de fluxo, **116**, 626
- showpos, manipulador de fluxo, 365, 626, **626**
- SIGABRT, 1009
- SIGFPE, 1009
- SIGILL, 1009
- SIGINT, 1009
- signal, função, **1009**
- SIGSEGV, 1009
- SIGTERM, 1009
- Silicon Graphics Standard Template Library Programmer's Guide, 956, 1062
- símbolo de acréscimo à saída (>>), **1003**
- símbolo de agregação, **103**, 103
- símbolo de decisão, **98**
- símbolo de redirecionamento de entrada (<), **1002**
- símbolo de redirecionamento de saída (>), **1002**
- símbolo especial, **669**
- símbolo, 704
- Simple, instrução, 835
- Simple, interpretador de, 845
- Simple, linguagem, 835
- Simpletron Machine Language (SML), 369, 799, 835, 836, 838, 843
- Simpletron Simulator, 369, 799, 836, 843, 844, 845
- Simpletron, posição da memória de, 845
- Simula, 8
- simulação de embaralhamento e distribuição de cartas, 336, 338, 340, 847, 850, 852
- simulação de supermercado, 833
- simulação de um baralho, 336
- simulação do jogo de dados *craps*, 197, 198, 199, 249
- simulação, 364
- Simulação: A tartaruga e a lebre, 361
- simulador de computador, 364
- simulador de voo, 594
- sin, função, 184
- sinais de pontuação, 352
- sinais definidos no cabeçalho <signal>, 1009
- sinais-padrão, 1009
- sinal alinhado à esquerda, 626
- sinal de adição (+) exibido na saída, 626
- sinal de adição (+) indicando visibilidade pública na UML, 400
- sinal de adição, + (UML), 59
- sinal de alerta interativo, 1009
- sinal de subtração (-) indicando visibilidade privada na UML, 67, 400
- sinal interativo, 1009
- sinal, **1009**, 1009
- sincronizar a operação de um istream e um ostream, 636
- sinônimo, 314, 316
- sintaxe, **29**
- sintetizador de fala, **1072**
- sistema bancário, 680
- sistema baseado em menus, 344
- sistema de contas a receber, 672, 699
- sistema de folha de pagamento, 669
- sistema de gerenciamento de banco de dados relacional (*relational database management system* — RDBMS), **730**
- sistema de gerenciamento de bancos de dados (*database management system* — DBMS), **671**
- sistema de numeração binário (base 2), 992
- sistema de numeração decimal (base 10), 626, 871, 992
- sistema de numero binário, 871
- sistema de pesquisa, 1066
- sistema de processamento de transação, **680**
- sistema de reservas de passagens aéreas, 304, 680
- sistema de software para controle e comando, 7
- sistema operacional, **4**, 6, 347, 674
- sistema, **45**
- sistemas baseados no Windows da Microsoft, 4
- sistemas operacionais de memória virtual, 8
- situações de missão crítica, 662
- size, atributo do elemento input, **1086**
- size, função de string, **682**
- size, função-membro da classe string, **705**
- size, função-membro de contêineres, 894
- size, função-membro de priority\_queue, **923**
- size, função-membro de queue, **921**
- size, função-membro de stack, **920**
- size, função-membro vector, 267, **288**
- size\_t, tipo, 327, 681
- size\_type, 895
- sizeof, operador, **327**, 329, 382, 429, 685, 687, 701, 800, 848, 873, 1024
  - utilizado para determinar tamanhos de tipo de dados padrão, 329
- skipws, manipulador de fluxo, 626
- smallest, 238
- SML, 362, 836, 838
- SML, código de operação, 362
- sobrecarga de operadores, **34**, 217, **451**, 612, 853
  - operadores de decremento, 479
  - operadores de incremento, 479
- sobrecarga de operadores, exercício final, 493
- sobrecarregar
  - +, 452, 454
  - +=, 453, 454
  - << e >>, 217, 218
  - definições de funções, 216
- função operadora de coerção, 468
- função-membro operator[], 468
- funções, 216, 217, 596, 597, 599
- operador [], 465
- operador +=, 481
- operador <=, 454
- operador binário <, 458
- operador de adição (+), 452
- operador de atribuição (=), 465, 467, 469, 477
- operador de atribuição de adição (+=), 480
- operador de chamada de função (), **478**, 479, 493
- operador de concatenação +=, 477
- operador de concatenação de string, 477
- operador de concatenação, 477
- operador de desigualdade, 465, 467
- operador de igualdade (==), 465, 467, 478
- operador de incremento pós-fixado, 479, 480, 481, 484
- operador de incremento prefixado, 480, 481, 484
- operador de incremento, 480
- operador de inserção de fluxo, 974
- operador de negação, 478
- operador de saída, 696
- operador de subscrito, 465, 468, 478, 494
- operador menor que, 478
- operador unário !, 457, 458
- operadores binários, 452, 458
- operadores de incremento prefixado e pós-fixado, 480
- operadores de inserção e extração de fluxo, 455, 456, 464, 465, 469, 480, 481
- operadores de pré-decremento e pós-decremento, 480
- operadores unários, 452, 457
- operadores, 217
- parênteses (), 953
- um operador como uma função não-friend, 454
- um operador de atribuição, 453
- uma função-membro, 382
- sobrescrever (XHTML), **1074**
- sobrescrever uma função, **557**
- software comercial, 540
- software confiável, 1012
- software de layout de página, 345
- software frágil, **525**
- software quebradiço, **525**
- software utilizado em grandes empresas varejistas, 7
- software, **2**, 3
- solicitação de navegador, 1083
- solicitação de terminação enviada para o programa, 1009
- solicitação de um serviço de um objeto, **57**
- solução de sobrecarga, 599
- Solução fatorial iterativa, 226, 227
- solução iterativa, **221**, 226
- solução recursiva, 226
- Solution Configurations, caixa de combinação, **1100**
- soma de dois inteiros, exercício de recursão, 227

- soma dos elementos de um array, 258
- somando inteiros com a instrução `for`, 147
- `sort`, algoritmo STL, 934, **936**
- `sort`, função da Standard Library, **783**
- `sort`, função-membro de `list`, **911**
- `sort_heap`, algoritmo STL, **946**
- `span`, atributo do elemento `col`, **1081**
- `span`, atributo, 1081
- `splice`, função-membro de `list`, **911**
- `spooler`, **817**
- spooling de impressão, **817**
- `sqrt`, função de arquivo de cabeçalho `<cmath>`, 184
- `square`, função, 190
- `srand( time( 0 ) )`, 195
- `srand`, função, **195**, 195
- `src`, atributo do elemento `img`, **1072**, 1072
- `<sstream>`, arquivo de cabeçalho, 191, **719**
- `stable_partition`, algoritmo STL, 901
- `stable_sort`, algoritmo STL, 949
- `Stack`, 599, 600
- `stack`, classe adaptadora, **919**, 919, 920
  - função `empty`, **920**
  - função `pop`, **919**
  - função `push`, **919**
  - função `size`, **920**
  - função `top`, **920**
- `Stack`, template de classe, 599, 600, 605, 815
  - com um objeto `List` composto, 817
  - definição, 814
- `Stack< double >`, 601, 605, 815
- `stack< int >`, 605
- `Stack< T >`, 600, 601, 603
- `<stack>`, arquivo de cabeçalho, 191, 895, **920**
- Standard Library (biblioteca-padrão), 183
  - arquivos de cabeçalho de contêiner, 894, 895
  - arquivos de cabeçalho, 191, 192, 1022
  - classe adaptadora `priority_queue`, 923
  - classe adaptadora `stack`, 920
  - classe `string`, 426, 484
  - classes contêineres, 893
  - classes de exceções, 662
  - hierarquia de exceções, 661
  - template de classe `deque`, 912
  - template de classe `list`, 908
  - template de classe `map`, 918
  - template de classe `multimap`, 917
  - template de classe `multiset`, 913
  - template de classe `set`, 916
  - template de classe `vector`, 903
  - templates de classe adaptadora `queue`, 922
- Standard Template Library (STL), 441, 579, 596, **891**
  - Programmer's Guide, 956, 1062
- Standard Template Library (STL), xxxii
- `static`, 1007, 1014
- `static`, especificador de classe de armazenamento, **200**
- `static`, especificador de linkagem, 966
- `static`, função-membro, 438, 440
- `static`, inicialização de array, 264
- `static`, membro de dados, 271, 274, **435**, 436, 437, 606
- `static`, membro de dados, monitorando o número de objetos de uma classe, 439
- `static`, membro, 436
- `static`, membros de dados, para poupar armazenamento, 436
- `static`, objeto local, 392, 394
- `static`, palavra-chave, 201
- `static`, variável local, 202, 264, 924
- `static_cast` (coerção de verificação dos tipos em tempo de compilação), 123, 164, 253
- `static_cast<int>`, 154
- `Status`, cabeçalho, **748**
- `std`, namespace, 704, 966
- `std::cin`, **31**, 33
- `std::cout`, 28
- `std::endl`, manipulador de fluxo, 33, 34
- `__STDC__`, constante simbólica predefinida, 1026
- `<stdexcept>`, arquivo de cabeçalho, 191, **646**, 661
- `<stdio.h>`, arquivo de cabeçalho, 192
- `<stdlib.h>`, arquivo de cabeçalho, 191
- `Step Into` (depurador), comando, **1106**
- `Step Out` (depurador), comando, **1106**
- `Step Over` (depurador), comando, **1106**
- `step`, comando de depurador, **1119**
- Stepanov, Alexander, 892, 956, 1062
- STL (Standard Template Library), **891**, 891, 892
- STL na programação genérica, 956, 1062
- STL, funções do contêiner, 894
- STL, referências, 956, 1062
- STL, software, 956
- STL, tipos de exceção da, 908
- STL, tutoriais, 955, 1062
- STL, xxxii
- `str`, função-membro da classe `ostringstream`, **719**
- `strcat`, função do arquivo de cabeçalho `<cstring>`, 348, **348**, 1007
- `strchr`, função do arquivo de cabeçalho `<cstring>`, **873**
- `strcmp`, função do arquivo de cabeçalho `<cstring>`, 348, **350**, 1007
- `strcpy`, função do arquivo de cabeçalho `<cstring>`, **347**, 348
- `strcspn`, função do arquivo de cabeçalho `<cstring>`, 873, **873**, 875
- `string` de caractere, **28**, 253, 263
- `string` de caracteres interna, 477
- `string` de consulta, **729**, 737, 738
- `string` é um ponteiro constante, 346
- `string find`, função-membro, 713
- `string` nula, 682
- `string` terminada por caractere nulo, 264, 336, 616, 717
- `string` tokenizada, 352, 353
- `string` vazia, **66**, **704**, 710
- `string`, 345
- `string`, 484
  - função `size`, **682**
- `string`, array de, **335**
- `String`, classe, 468, 469
  - definição da, com sobrecarga de operadores, 469
- definições de função-membro e função
  - `friend` da, 471
  - programa de teste da, 474
- `string`, classe, **60**, 426, 442, 451, 484, 487, 704, 705
  - construtor da, 704
  - da Standard Library, 191
  - função `data`, **682**
  - função-membro `insert`, 716, **717**
  - função-membro `length`, **80**
  - função-membro `substr`, **80**, 487
- `string`, comprimento de uma, 353
- `string`, literal de, **28**, 33, 263, 264, **346**
- `string`, manipulação de, 183, 841
- `string`, objeto
  - `string` vazia, **66**
  - valor inicial, **66**
- `string`, tipo de dados, 442
- `<string.h>`, arquivo de cabeçalho, 191
- `string::npos` (fim da string), **712**, 713, 714
- `<string>`, arquivo de cabeçalho, **60**, 72, 191, 484, 704
- strings baseadas em ponteiro, 346
- strings como objetos completos com todos os recursos, 345
- strings de caracteres, **28**
- strings, atribuição de, 705
- strings, comparando, 707
- strings, concatenação de, 705
- strings, cópia de, 333
  - utilizando a notação de array e a notação de ponteiro, 335
- strings, processamento de, 312
- `strlen`, função, 348, **353**
- `strncat`, função, 348, **349**
- `strncmp`, função, 348, **350**
- `strncpy`, função, **347**, 348
- `strong` (XHTML), elemento, **1068**
- Stroustrup, Bjarne, 6, 8, 21, 596, 645, 1060
- `strpbrk`, função, 873, **873**
- `strrchr`, função, 873, **873**
- `strspn`, função, 873, **874**
- `strstr`, função, 873, **874**
- `strtod`, função, 869, **870**, 870, 871
- `strtok`, função, 348, **352**
- `strtol`, função, 869, **871**, 871
- `strtoul`, função, 869, **871**
- `struct`, 799, **847**, 1013, 1014
- `sub` (XHTML), elemento, **1074**
- subárvore direita, **820**, 825, 827, 834
- subárvore esquerda, **820**, 825, 827, 834
- subclasse, **502**
- sublinhado ( `_` ), 32
- `submit` (type), valor do atributo, 742
- subobjeto de classe básica, 978
- subproblema, 221
- subscrver (XHTML), **1074**
- subscrver com um ponteiro e um deslocamento, 333
- subscrver por meio de um vector, 907
- subscrito 0 (zero), 251
- subscrito de array duplo, 493
- subscrito de array fora do intervalo, 650
- subscrito de coluna, 280

subscrito de linha, 280  
 subscrito fora do intervalo, 908  
 subscrito, **251**  
 subscrito, acesso indexado utilizando, 911  
 subscrito, operador de, 912  
 subscrito, verificação de intervalo de, 441  
 substantivo, 8  
 substantivos compostos no documento de requisitos, 82, 123  
 substantivos em uma especificação de sistema, 18  
 substantivos na declaração do problema, 139  
 substituição de texto, 1023  
 substituir o operador == por =, 164, 165  
 substr, função-membro de string, **80**, 487, **709**, 709, 742  
 substring de uma string, 709  
 substring, 478  
 substring, comprimento da, 478  
 subtração de ponteiro, 330  
 subtração, 3, 35, 36  
 subtrair um inteiro de um ponteiro, 330  
 subtrair um ponteiro de outro, 330  
 summary, atributo do elemento table, **1077**  
 sup (XHTML), elemento, **1074**  
 superclasse, **502**  
 supercomputador, **3**  
 swap, algoritmo STL, 901, 937, **937**  
 swap, função-membro da classe string, **709**, 709, 710  
 swap, função-membro de contêineres, 894  
 swap, função-membro de list, **911**  
 swap\_ranges, algoritmo STL, 901, 937, **938**  
 switch, instrução de seleção múltipla, 151, 156, 167  
 switch, instrução, 696  
 switch, lógica, 158, 563, 592

## T

Tab, tecla, 29  
 tabela de função virtual (vtable), **579**  
 tabela de símbolos, 838, 839, 841, 842  
 tabela de valores, **279**  
 tabela-verdade, **161**  
   operador ! (NÃO lógico), 162  
   operador && (E lógico), 162  
   operador || (OU lógico), 162  
 table (XHTML), elemento, **1077**  
 table row (tabela XHTML), **1081**  
 tabulação \t, sequência de escape de, 98, 156  
 tabulação horizontal (' \t '), 29, 864, 866  
 tabulação vertical (' \v '), 864, 866  
 tabulação, parada de, 29  
 tabuleiro, padrão de, 53, 138  
 tag final (XHTML), **1065**  
 tag inicial (XHTML), **1065**  
 tamanho de um array, 264, 327, 328  
 tamanho de uma estrutura, 848  
 tamanho de uma string, 710  
 tamanho de uma variável, 34, 200  
 tamanho de variável, 34  
 tamanho fixo de palavras, 441  
 tamanho máximo de uma string, **710**  
 tamanhos de inteiro menores, 159  
 tamanhos de tipo de dados padrão, 329  
 tamanhos dos tipos de dados predefinidos, 702  
 tan, função, 184  
 tangente trigonométrica, 184  
 tangente, 184  
 tarefa, **4**  
 tartaruga e a lebre, A, 361  
 taxa de juros, 147, 178  
 tbody (XHTML), elemento, **1081**  
 td (XHTML), elemento, **1081**  
 teclado numérico, 46  
 teclado, 3, 4, 10, 33, 154, 156, 362, 363, 613, 614, 671, 1002  
 tela de vídeo, 613, 615  
 tela, 3, 4, 10, 27, 28  
 telefone, 138  
 tellg, função de istream, **677**  
 tellp, função de ostream, **677**  
 Temperaturas Celsius e Fahrenheit, exercício, 244  
 template de classe de listas vinculadas, 845  
 template de classe, **596**, 596, 599, 704, 802, 821, 824  
   definição de, 599, 600  
   escopo do, 600  
   especialização de, **596**, 599, 600  
   especialização explícita do, **605**  
   Stack, 600, 602  
 template de função, 218, **219**, 219, **596**, 596, 597, 598, 599  
   definições de, **597**  
   especializações de, **219**, **596**  
   max, 249  
   min, 249  
 template, **596**, 799, 802, 813, 1023  
 template, definição de, 219  
 template, palavra-chave, 219, **597**  
 templates de função, sobrecarregando, 599  
 templates de processamento de arquivo, 615  
 templates e amigos, 606  
 templates e herança, 606  
 tempo de execução constante, **782**  
 tempo de execução de pior cenário para um algoritmo, 782  
 tempo de execução linear, **782**  
 tempo de execução logarítmico, **785**  
 tempo de execução quadrático, **782**  
 tentando chamar uma função multiplamente herdada polimorficamente, 978  
 tentando modificar dados por meio de um ponteiro não constante para dados constantes, 322  
 tentando modificar um ponteiro constante para dados constantes, 325  
 tentando modificar um ponteiro constante para dados não constantes, 323  
 Tentativa errônea de inicializar uma constante de um tipo de dados predefinido por atribuição, 419  
 terceiro refinamento, 338  
 terminação de programa, 392, 394, 1007  
 terminação malsucedida, 1007  
 terminador de instrução (:), **28**  
 terminal, 4  
 terminar a execução, 442

terminar com sucesso, 29  
 terminar na chave direita (}) de um bloco, 202  
 terminar normalmente, 674  
 terminar um loop, 110  
 terminar um programa, 658, 1007  
 terminar uma estrutura de repetição, 1012  
 terminate, função, **651**, 653  
 término, fase de, 109  
 test, 950  
 Testando estados de erro, 635  
 testar bits de estado depois de uma operação de E/S, 617  
 testar, 11  
 teste de caracteres, 191  
 teste de terminação, 226  
 teste de validade, **78**  
 text (type), valor do atributo, 742  
 text/html, tipo MIME, **733**  
 text/plain, tipo MIME, **733**  
 textarea, elemento (XHTML), **1086**  
 textarea, elemento XHTML, 742  
 textarea, elemento, 1086  
 texto formatado, 680  
 texto substituto, 1023, 1025  
   para uma macro ou constante simbólica, **1023**, 1024  
 texto, análise de, 371  
 texto, navegador baseado em, **1072**  
 tfoot (XHTML), elemento, **1081**  
 th (coluna de cabeçalho da tabela), elemento, 1081  
 thread (XHTML), elemento, **1081**  
 this, ponteiro, 426, **429**, 429, 430, 431, 440, 454, 467  
 this, ponteiro, utilizado explicitamente, 430  
 Thompson, Ken, 6  
 throughput, **4**  
 throw(), especificação de exceção, 653  
 throw, lista, **653**  
 throw, palavra-chave, **650**  
 TicTacToe, classe, exercício, 410  
 tie, 636  
 til (~), caractere, 391  
 Time, classe, 409  
 Time, definição da classe, 377  
 time, função, 195, **735**  
 <time.h>, arquivo de cabeçalho, 191  
 \_\_TIME\_\_, constante simbólica predefinida, 1026  
 time\_t, tipo, **735**  
 tipo de iterador 'mais fraco', 897  
 tipo de retorno, **59**, 189  
   na UML, 229, 233  
   void, **59**, 66  
 tipo de sequência, **1077**  
 tipo de uma variável, 34, 200  
 tipo definido pelo usuário, **59**, 199, 468  
 tipo do ponteiro this, 430  
 tipo fundamental, **32**  
 'tipo mais alto', 189  
 'tipo mais baixo', 189, 190  
 tipo parametrizado, **599**, 609  
 tipo predefinido, **32**, 440, 441  
 tipo primitivo, **32**  
   promoção, 115



tipos de dados abstratos (*abstract data types* — ADTs), 440, 441, 442

tipos de dados agregados, 322, 847

tipos de dados

bool, 98

char, 154, 190

double, 111, 115, 147, 148

float, 111, 190

int, 31

long double, 190

long int, 158, 190

long, 158

na UML, 62

short int, 158

short, 158

unsigned char, 190

unsigned int, 190, 195, 327

unsigned long int, 190

unsigned long, 190

unsigned short int, 190

unsigned short, 190

unsigned, 195

tipos de dados, 440, 441

tipos definidos pelo usuário, 17, 18

title (<title>...</title>), elemento

XHTML, 732

title (XHTML), elemento, 1066

título de um documento, 1065

títulos de coluna, 253

TLD (*top-level domain* — domínio de nível mais alto), 730

token, 348, 352, 841, 842

tokenizar strings, 347, 352

tokenizar uma frase em palavras separadas, 833

tolower, 864, 866

top, função-membro de `priority_queue`, 923

top, função-membro de `stack`, 920

topo, 109

Torres de Hanói, 246, 246, 247

Torres de Hanói, exercício de recursão, 227

total, 104, 105, 109, 110, 201

toupper, função (<ctype>), 321, 864, 866, 866

tr (XHTML), elemento, 1081

</tr> (tag final XHTML), 737

<tr> (tag inicial XHTML), 737

trabalhador comissionado, 179

trabalhador por produção, 179

trabalho, 4

trace, tipo de solicitação HTTP, 729

tradução, 5

transação, 699

Transaction, classe (estudo de caso ATM), 586, 587, 588, 589, 590, 591

transferência de controle, 95

transform, algoritmo STL, 901, 932, 934

transição entre estados na UML, 169

transição, 96

transmitir com segurança, 138

trapezóide, 503

tratamento de exceções, 191, 645

travel.html, 749

Tree, template de classe, 821, 822

Tree< int >, 821

TreeNode, template de classe, 821, 822

triângulo retângulo, 179

triângulo reto, 138

tripleByReference, 249

tripleByValue, 249

triplos de Pitágoras, 179

troca de valores, 278, 324

trocando strings, 709

true, 98, 98, 99, 100, 142

truncado(a)(s), 35, 108, 115, 426, 430, 673

truncar a parte fracionária de um double, 189

try, bloco, 647, 651, 654

try, palavra-chave, 647

turing machine, 95

type, atributo do elemento ol, 1077

type, atributo, 1077, 1084, 1085

type info, classe, 585, 585

typedef de iterador, 899

typedef fstream, 615

typedef ifstream, 615

typedef iostream, 614

typedef istream, 614

typedef ostream, 615

typedef ostream, 614

typedef, palavra-chave, 614, 704, 719, 849,

894, 895, 913, 915, 918

typedefs em contêineres de primeira classe, 895

typeid, operador, 585, 661

<typeinfo>, arquivo de cabeçalho, 191, 585

typename, palavra-chave, 219, 597, 597

## U

U, sufixo de inteiro, 1009

u, sufixo de inteiro, 1009

ul (XHTML), elemento, 1075

ul, elemento, 1075

UL, sufixo de inteiro, 1009

ul, sufixo de inteiro, 1009

último a entrar, primeiro a sair (*last-in, first-out*

– LIFO), 204, 893

estrutura de dados, 813, 919

ordem, 599, 602

‘ultrapassar’ uma das extremidades de um array,

458

UML (Unified Modeling Language), 16, 17, 18,

41, 45, 59, 83, 96, 125, 126, 586

diagrama da, 45

UML Partners, 19

UML

aspas francesas (« e »), 70

atributo, 59

círculo pequeno, 96

círculo sólido, 96

condição de guarda, 98, 99, 103, 104

construtor em um diagrama de classes, 70

decisão, 99

diagrama de atividades, 96, 96, 99, 100, 103,

104, 145, 146

diagrama de classes, 59

estado de ação, 96, 96

estado final, 96

estado inicial, 96

expressão de ação, 96, 99, 104

linha pontilhada, 96

losango, 96, 98

nota, 96

operação pública, 59

seta de transição, 96, 98, 99, 103, 104

seta, 96

símbolo de agregação, 103

símbolo de decisão, 98

sinal de adição (+), 59

sinal de subtração (–), 67

tipo, String, 62

tipos de dados, 62

transição, 96

UML, diagrama de atividades

círculo sólido (para representar um estado inicial) na UML, 170

círculo sólido dentro de um círculo vazado (para representar o fim de uma atividade) na UML, 170

UML, diagrama de atividades, 96

UML, diagrama de casos de uso na, 44, 45

ator, 45

UML, diagrama de classes

compartimento de atributos, 125

compartimento de operação, 229

construtor, 70

UML, diagrama de estados

círculo sólido (para representar um estado inicial) na UML, 169

retângulo arredondado (para representar um estado) na UML, 169

UML, diagrama de seqüências

ativação, 295

linha da vida, 295

seta, 295

unconditional branch, instrução, 842

#undef, diretiva de pré-processador, 1024, 1024, 1026

underflow\_error, exceção, 662

unexpected, função, 653, 661

único ponto de entrada, 165

único ponto de saída, 165

Unicode®, 613, 669, 704

unidade central de processamento (*central processing unit* – CPU), 3

unidade de aritmética e lógica (*arithmetic and logic unit* – ALU), 3

unidade de armazenamento endereçável, 863

unidade de armazenamento secundária, 4

unidade de armazenamento, 863

unidade de compilação, 966

unidade de disco, 613

unidade de entrada, 3

unidade de memória, 3

unidade de processamento, 3

unidade de saída, 3

unidade lógica, 3

unidades ‘independentes’, 4

unidades, posição das, 992

Unified Modeling Language (UML), 16, 17, 18,

41, 45, 83, 96, 125, 126, 586

Uniform Resource Locator (URL), 728

union anônima, 1014

union sem construtor, 1014

union, construtor, 1014

union, 1012, 1013, 1014

anônima, **1014**  
 union, funções, não podem ser declaradas virtual, **1014**  
 unique, algoritmo STL, 901, 938, **938**  
 unique, função-membro de list, **911**  
 unique\_copy, algoritmo STL, 901, 940, **940**  
 UNIX, 4, 6, 8, 623, 674, 731, 1005, 1007, 1009  
 UNIX, linha de comando, 1002  
 unsigned char, tipo de dados, 190  
 unsigned int, tipo de dados, 190, 195, 327  
 unsigned long int, tipo de dados, 190, 222  
 unsigned long, tipo de dados, 190, 222, 871, 1009  
 unsigned short int, tipo de dados, 190  
 unsigned short, tipo de dados, 190  
 unsigned, tipo de dados, 190, 195, 1009  
 upper\_bound, algoritmo STL, 943, **943**  
 upper\_bound, função de contêiner associativo, **915**  
 uppercase, manipulador de fluxo, 626, 630, 631  
 URL (Uniform Resource Locator), **728**, 732  
 us (domínio de nível mais alto), 731  
 USENIX C++ Conference, 596  
 userdata.txt, 762  
 using namespace, diretiva, 966  
 using, declaração, **38**, 966  
 using, diretiva, 966  
 <utility>, arquivo de cabeçalho, 192  
 utilização do ponteiro this explicitamente, 430  
 utilizando a função swap para trocar duas strings, 710  
 utilizando a função-membro fill e o manipulador de fluxo setfill para alterar o caractere de preenchimento dos campos maiores que os valores sendo impressos, 629  
 utilizando argumentos de linha de comando, 1006  
 Utilizando as classes básicas virtual, 980  
 utilizando as funções exit e atexit, 1008  
 utilizando funções de template, 598  
 utilizando funções Standard Library para realizar um heapsort, 946  
 utilizando goto, 1013  
 utilizando listas de argumentos de comprimento variável, 1004  
 utilizando o manipulador de fluxo uppercase, 632  
 utilizando tratamento de sinal, 1010  
 utilizando um array em vez de switch, 260  
 utilizando um iterador para gerar a saída de uma string, 718  
 Utilizando um membro de dados static para manter uma contagem do número de objetos de uma classe, 437  
 utilizando um objeto ostream dinamicamente alocado, 720  
 utilizando um template de função, 219  
 utilizando uma union anônima, 1015

## V

va\_arg, 1003, **1003**  
 va\_end, 1003, **1003**  
 va\_list, 1003, **1003**  
 va\_start, 1003, **1003**

validação, **78**, 719  
 valign, atributo do elemento th, **1081**  
 valor 'lixo', **107**  
 valor absoluto, 184  
 valor de ponto fixo, 149  
 valor de ponto flutuante de notação científica, 631  
 valor de sentinela, **109**, 110, 113, 114, 155  
 valor de sinal, **109**  
 valor de um atributo, **1066**  
 valor de um elemento de array, **252**  
 valor de uma variável, 34, 200  
 valor fictício, **109**  
 valor final de uma variável de controle, **141**, 145  
 valor indefinido, **107**  
 valor inicial de um atributo, 125, 126  
 valor inicial de uma variável de controle, **141**, 143  
 valor por extenso, 373  
 valor posicional, 138, 993  
 valor temporário, 115, 189  
 valor, **33**  
 valorDeDeslocamento, **195**  
 valores de face das cartas, 336  
 valores de nó duplicados, 821  
 valores de ponteiro como inteiros hexadecimais, 314  
 valores mapeados, **913**  
 valores posicionais no sistema de numeração decimal, 993  
 value, atributo do elemento input, **1084**  
 value\_type, 894, 895, 918  
 variável automática, 814  
 variável constante visível no depurador, 1023  
 variável constante, **255**, 255, 256, 258  
 variável contadora, 107  
 variável de ambiente, **737**  
     CONTENT\_LENGTH, **745**  
     HTTP\_COOKIE, **753**  
     QUERY\_STRING, **737**  
 variável de classe, **271**  
 variável de controle, 143  
     nome, 144  
 variável de escopo de classe é ocultada, 382  
 variável de leitura, **255**  
 variável de ponteiro, 659  
 variável global, **201**, 202, 204, 215, 267, 966, 968, 1005  
 variável local automática, 200, 202, 212, 213  
 variável local, **62**, 200, 201, 202, 204, 1014  
 variável não inicializada, 107  
 variável, **31**  
 VAX (VMS), 674  
 vazamento de recurso, 654, 663  
 vector, classe, 288, 484  
     função capacity, **903**, 904  
     função push\_back, **903**  
     função rbegin, **905**  
     função rend, **905**  
 vector, funções de manipulação de elemento do template de classe, 906  
 vector, template de classe, 902, 903  
 <vector>, arquivo de cabeçalho, 191  
 <vector>, arquivo de cabeçalho, 288, 895

verbos em uma especificação de sistema, 17  
 verbos na declaração de um problema, 139  
 verificação de erros, 183  
 verificação de intervalos, 458, **705**, 901  
 verificação de limite, 263  
 verificação de sintaxe, 844  
 verificação de tipos, 596, 1023, 1024  
 vi, editor de textos, 10, 1064  
 viewcart.cgi, 767  
 vinculação dinâmica, 547, **557**, **558**, 579, 582  
 vinculação estática, **558**  
 vinculação tardia, **558**  
 vincular ao código-objeto de uma classe, 382  
 vincular, 10, 1007  
 vínculo (ou link), **800**, 800, 801, 820, **1068**  
 vínculos de ponteiro, **800**  
 violação de acesso de memória, xxxiii, 347, 892, 893  
 violação de acesso, xxxiii, 347, 892, 893  
 violação de segmentação, **1009**  
 virtual pura, função, **563**, 579  
 virtual what, função da classe exception, **646**, 646, 657  
 virtual, chamada de função, 581  
 virtual, chamada de função, ilustração, 580  
 virtual, classe básica, 978, **980**  
 virtual, destrutor, **585**, 585  
 virtual, função, 547, **557**, 557, 579, 581, 895, 924, 978, 1014  
 virtual, herança, 964, 980  
 visibilidade na UML, **399**  
 Visual Basic .NET, **8**  
 Visual C++ .NET, **8**  
 Visual C++, home page, 20, 1061  
 Visual Studio .NET  
     caixa Quick Info, **1102**  
 visualizando a recursão, 227  
 VMS, 1005  
 void \*, 331, 332, 877, 878  
 void, palavra-chave, **59**, 66  
 void, tipo de retorno, 189  
 volatile, qualificador, **964**, **1009**  
 'voltas' ou 'reinícios' ao incrementar o último dia do mês, função para lidar com as, 484  
 volume de um cubo, 209  
 vtable, **579**, 579, 580, 581, 582  
 vtable, ponteiro, 581, 582

## W

Watch (depurador Visual C++ .NET), janela, **1103**, 1103, 1104  
 watch, comando de depurador, **1121**  
 wchar\_t, tipo de caracteres, **613**, **704**  
 webstore.ansi.org/ansidocstore/default.asp, 11  
 what, função da classe runtime\_error, 650  
 while, instrução de repetição, **97**, 103, 114, 141, 142, 150, 169  
 width, atributo do elemento img, **1072**  
 width, atributo do elemento table, **1077**  
 width, função-membro da classe ios\_base, **622**, 623  
 Windows 2000, 731  
 Windows, 155, 1002, 1009

Withdrawal, classe (estudo de caso ATM), 83, 84, 85, 124, 170, 229, 293, 294, 295, 296, 401, 404, 585, 586, 587, 588, 589, 590

WordPad, 1064

World Wide Web, **5**

World Wide Web, recursos STL na, 955

write, função de ostream, 615, 620, 681, 685  
www.accu.informika.ru/resources/  
public/terse/cpp.htm, 21

www.acm.org/crossroads/xrds3-2/ovp32.html, 21

www.borland.com, 10

www.borland.com/bcppbuilder, 21

www.codearchive.com/list.php?go=0708, 21

www.compilers.net, 21

www.cuj.com, 21

www.deitel.com, 2, 20, 1068

www.deitel.com/newsletter/subscribe.html, 3, 20

www.devx.com, 21

www.forum.nokia.com/main/0,6566,050\_20,00.html, 21

www.gametutorials.com/Tutorials/GT/GT\_Pg1.htm, 21

www.hal9k.com/cug, 21

www.jasc.com, 1070

www.kai.com/C\_plus\_plus, 21

www.mathtools.net/C\_C\_/Games/, 21

www.metrowerks.com, 10

www.msdn.microsoft.com/visualc/, 10

www.omg.org, 19

www.prenhall.com/deitel, 20

www.research.att.com/~bs/homepage.html, 21

www.symbian.com/developer/development/cppdev.html, 21

www.thefreecountry.com/developercity/ccompilers.shtml, 20

www.uml.org, 19

www.unicode.org, 704

www.w3.org/markup, 1064

www.w3.org/TR/xhtml1, 1092

www.w3schools.com/xhtml/default.asp, 1092

www.xhtml.org, 1092

## X

XHTML (Extensible HyperText Markup Language), xxxi, 728, 732, **1064**

comentário, **1065**

documento, **1064**

elementos, **1065**

formulário, 741, 1083

tabela, **1077**

XHTML 1.1 Recommendation, 1092

xor, palavra-chave de operador, **969**

xor\_eq, palavra-chave de operador, **969**

## Z

zero-ésimo elemento, **251**

zeros e uns, 669

zeros finais, 626, 627



## Utilizando o CD-ROM

A interface para o conteúdo deste CD é projetada para iniciar automaticamente pelo arquivo `AUTORUN.EXE`. Se uma tela de abertura não abrir automaticamente ao inserir o CD no computador, dê um clique duplo no arquivo `welcome.htm` para carregar o CD ou consulte o arquivo `Leiam.txt` no CD.

## Conteúdo do CD-ROM

- Downloads de software: links para compiladores e ferramentas de desenvolvimento em C++ livres ou gratuitas
- Exemplos
- Recursos na Web

## Requisitos de software e hardware

- Processador Pentium II 450 MHz (mínimo) ou mais rápido
- Microsoft® Windows® Server 2003, XP Professional, XP Home Edition, XP Media Center Edition, XP Tablet PC Edition, 2000 Professional (SP3 ou superior necessário para instalação), 2000 Server (SP3 ou superior necessário para instalação), ou
- Um dos seguintes distribuidores do Linux: Fedora Core 3 (ex-Red Hat Linux), Mandrakelinux 10.1 Official, Red Hat 9.0, SUSE LINUX Professional 9.2 ou Turbolinux 10 Desktop
- Além da quantidade mínima de RAM requerida pelos itens listados acima, não há nenhum requisito adicional de RAM para este CD-ROM.
- Unidade de CD-ROM
- Conexão com a Internet
- Ambiente C++
- Navegador Web, Adobe® Acrobat® Reader® e um utilitário de descompactação Zip

# COMO ++ PROGRAMAR

— 5ª Edição —

A melhor e mais respeitada introdução a C++, programação orientada a objetos (POO) e projeto orientado a objetos (OOD) com UML™ 2 utilizando o Deitel® Live-Code.

O C++ é uma das mais populares linguagens de programação orientada a objetos. Esta nova edição do livro-texto de C++ mais utilizado no mundo baseia-se na introdução a classes e objetos.

“Excelente introdução a classes e objetos. A combinação de exemplos de código ativo e figuras detalhadas fornece uma visão única dos conceitos de C++.” — **Earl LaBatt, Universidade de New Hampshire**

“Esta nova edição indica um passo significativo na evolução pedagógica da série Deitel em C++. A perspectiva de antecipar o estudo de classes e objetos é consistente com o pensamento atual sobre a educação dos desenvolvedores de software.” — **Ric Heishman, Northern Virginia Community College**

“Eu sempre me impressiono com a capacidade dos Deitel de explicar claramente conceitos e ideias, o que permite ao aluno adquirir um bom entendimento da linguagem e do desenvolvimento de software.”

— **Karen Arlien, Bismarck State College**

“O livro é abrangente, correto e claro como cristal. Nenhum outro texto aborda tão cuidadosamente as complexidades dessa poderosa linguagem.” — **James Huddleston, consultor independente**

“Este livro é um dos melhores de sua categoria. É uma excelente cobertura de C++ que antecipa a introdução a objetos e mesmo assim permanece acessível para os iniciantes. A apresentação baseada em exemplos é enriquecida pelo estudo de caso opcional OOD/UML ATM que contextualiza o material em um projeto de engenharia de software contínuo.”

— **Gavin Osborne, Saskatchewan Institute**

“Apresentar a UML aos alunos é uma excelente ideia.” — **Raymond Stephenson, Microsoft**

“Excelente cobertura sobre o polimorfismo. Gostei especialmente da maneira de o livro explicar como o compilador implementa o polimorfismo 'sob o capô'. Gostaria de ter tido uma apresentação tão clara das estruturas de dados quando eu era aluno.” — **Ed James-Beckham, Borland**

“Adorei o estudo de caso opcional de ATM! Pode ser a experiência fundamental que combina os blocos de construção da linguagem e os conceitos de engenharia de software em um exemplo extremamente relevante.”

— **Karen Arlien, Bismarck State College**

“Inclui uma ótima introdução à pesquisa e classificação, especialmente com a introdução à notação O (sem ser demasiadamente teórica sobre o assunto). Os exercícios são especialmente bons.”

— **Robert Myers, Florida State University**

**C++ como programar** oferece uma cobertura abrangente sobre a programação orientada a objetos em C++, incluindo vários estudos de caso integrados de grande relevância: a classe GradeBook (capítulos 3 – 7), a classe Time (três seções do Capítulo 9), a classe Employee (capítulos 12 – 13) e o opcional OOD/UML™ 2 ATM System (capítulos 1 – 7, 9 e 13).

O **dr. Harvey M. Deitel** e **Paul J. Deitel** são os fundadores da **Deitel & Associates, Inc.**, uma organização internacionalmente reconhecida de treinamento corporativo e criação de conteúdo especializada em C++, Java™, C, C#, Visual Basic® .NET, Visual C++® .NET, XML, Python, Perl, Internet, Web e tecnologias de objeto. Os Deitel são autores de muitos livros-texto sobre linguagem de programação, líderes de vendas internacionalmente, entre eles *Java como programar*, 6 ed., *Sistemas operacionais*, 3 ed., *Internet & World Wide Web como programar*, 3 ed. e *C How to Program*, 4 ed.

Contato com os autores: [deitel@deitel.com](mailto:deitel@deitel.com)

Para informações sobre a série de treinamentos corporativos em **DEITEL® DIVE INTO™ SERIES**, oferecido em todo o mundo, e para assinar o boletim de correio eletrônico **DEITEL® BUZZ ON-LINE**, visite: [www.deitel.com](http://www.deitel.com)



[sv.pearson.com.br](http://sv.pearson.com.br)

A Sala Virtual oferece apresentações em PowerPoint traduzidas e manual de soluções (em inglês) para professor; para o estudante, código-fonte dos exemplos do livro e exercícios de múltipla escolha (em inglês).

[www.pearson.com.br](http://www.pearson.com.br)

ISBN 978-85-7605-056-8

